NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

IMPLEMENTATION OF A LANGUAGE TRANSLATOR
FOR THE
COMPUTER AIDED PROTOTYPING SYSTEM

by

Charles E. Altizer

December 1988

Thesis Advisor: Luqi

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited
2b Declassification/Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (If Applicable): 37	7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (city, state, and ZIP code): Monterey, CA 93943-5000	7b Address (city, state, and ZIP code): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number
8c Address (city, state, and ZIP code)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

11 Title (Include Security Classification): IMPLEMENTATION OF A LANGUAGE TRANSLATOR FOR THE COMPUTER AIDED PROTOTYPING SYSTEM

12 Personal Author(s): Altizer, Charles E.

13a Type of Report	13b Time Covered	14 Date of Report (year, month,day)	15 Page Count
Master's Thesis	From To	1988 December	162

16 Supplementary Notation: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)
Field	Group	Subgroup	Ada, Rapid Prototyping, Software Reusability, Computer Aided Prototyping System, Prototype System Description Language, CAPS, PSDL, Kodiyak

19 Abstract (continue on reverse if necessary and identify by block number

Rapid prototyping is a method of software system development that is gaining much support presently. Rapid prototyping allows the designer to quickly produce a model of a system or part of a system which the user can see and thus verify if his requirements have been met. The prototype specifications can then be efficiently converted to an accurate set of program specifications that the programmers can implement as a final working system. The computer aided prototyping system (CAPS) is a rapid prototyping system that will automate many of the processes of prototyping such as code generation of prototype modules and searching for reusable components.

One of the many components of CAPS is a language translator which translates a prototype specification written in the Prototype System Description Language (PSDL) into a set of Ada procedures and packages. The Ada procedures and packages, when executed in proper order, will effectively execute the prototype. This thesis demonstrates an implementation of the translator component of the CAPS. An attribute grammar tool, Kodiyak, is used to build a translator which implements the major constructs of PDSL and produces Ada code to implement PSDL operators according to their control constraints.

20 Distribution/Availability of Abstract	21 Abstract Security Classification
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	Unclassified
22a Name of Responsible Individual: Luqi	22b Telephone (Include Area code): (408) 646-2735; 22c Office Symbol: 52LQ

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted

All other editions are obsolete

security classification of this page: Unclassified

Approved for public release; distribution is unlimited.

Implementation Of A Language Translator
For The
Computer Aided Prototyping System

by

Charles Edwin Altizer
Lieutenant, United States Navy
B.S., University of Florida, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

Author: _____
Charles Edwin Altizer

Approved By: _____
Luqi, Thesis Advisor

_____
Valdis Berzins, Second Reader

_____
Robert B. McGhee, Chairman,
Department of Computer Science

_____
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

Rapid prototyping is a method of software system development that is gaining much support presently. Rapid prototyping allows the designer to quickly produce a model of a system or part of a system which the user can see and thus verify if his requirements have been met. The prototype specifications can then be efficiently converted to an accurate set of program specifications that the programmers can implement as a final working system. The computer aided prototyping system (CAPS) is a rapid prototyping system that will automate many of the processes of prototyping such as code generation of prototype modules and searching for reusable components.

One of the many components of CAPS is a language translator which translates a prototype specification written in the Prototype System Description Language (PSDL) into a set of Ada procedures and packages. The Ada procedures and packages, when executed in proper order, will effectively execute the prototype. This thesis demonstrates an implementation of the translator component of the CAPS. An attribute grammar tool, Kodiyak, is used to build a translator which implements the major constructs of PSDL and produces Ada code to implement PSDL operators according to their control constraints.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Rapid prototyping is the process of producing a model of a software system to be evaluated by the system designer and the system user. It is becoming a widely accepted method for software systems design. In a conventional systems development cycle the system designer and the user work together to determine a set of user requirements and goals for the complete system. These are eventually converted into a set of specifications which can be implemented in a particular programming language. If all goes well, the system is implemented according to the specifications and is thus a true implementation of the user's requirements and goals.

Unfortunately, systems development is much more difficult than this. Rarely is it the case that the designer successfully captures the user's true requirements. Ordinarily, the designer is knowledgeable of computer systems and programming but not of the user's specialty. The same holds true of the user, he knows his application and how it should operate but he knows very little of computers and their capabilities. Consequently, it is difficult to capture the true requirements of the system for several reasons including:

- The designer is prone to misinterpretation of the user's goals because they are unfamiliar to him.
- The user often states his goals and requirements in overly general terms.
- The user may completely omit other goals and requirements simply because they are obvious to him while they are not obvious to the designer.
- Because of his unfamiliarity with computer systems and their capabilities, the user may not be aware of some features that are available until later in the project as he becomes more familiar with the system. [Ref. 1]

Whatever the reason, the end result is an incomplete set of requirements that the designer must formalize into a set of specifications to be implemented in software. Programmers implement the system and it is presented to the user but rejected because it does not functionally represent what the user thought he wanted in the first place. The user points out the discrepancies and then the designer attempts to

re-analyze the user's requirements, form an updated set of specifications, then the system is re-implemented, and presented again to the user. This software development loop continues until a version of the system is finally accepted by the user.

Software systems today are becoming extremely large and complex. Utilizing methods such as that described above to construct modern software systems results in unreasonably long development life cycles and unreliable systems. Productivity must be improved to produce such systems in a reasonable amount of time today. A method must be found that will allow the user and designer to produce a complete and accurate set of requirements prior to code generation of the actual system. Rapid prototyping is such a method.

The rapid prototyping loop shown in Figure 1 replaces the conventional software development process. In rapid prototyping the user and designer still perform a requirements analysis but now a software model of the system, or part of the system, is quickly assembled and executed for the user instead of a complete system. The model is constructed knowing that it is probably going to be altered over and over, however new models can be produced quickly so the cost of producing the prototype is relatively cheap.

The user can evaluate the prototype and point out any discrepancies. The discrepancies are soon corrected and a new prototype is available for the user to evaluate. This prototype loop continues until the user is satisfied with a particular version [Ref. 2]. The final prototype can then be used to generate a set of specifications that accurately reflect the user's goals and requirements. Since the programmer is presented with an accurate set of specifications, he implements the system once and wastes little time performing code corrections as before. Thus productivity and reliability are greatly enhanced.

The computer aided prototyping system (CAPS) is a prototyping system intended to improve current prototyping methods by automating time consuming tasks in conventional prototyping such as turning specifications into prototypes, modifying prototypes, and searching for reusable components. The components which make up the CAPS system include a specification language, an execution support system, a

rewrite system, a syntax-directed editor, a software base, a design database, and a design management system.

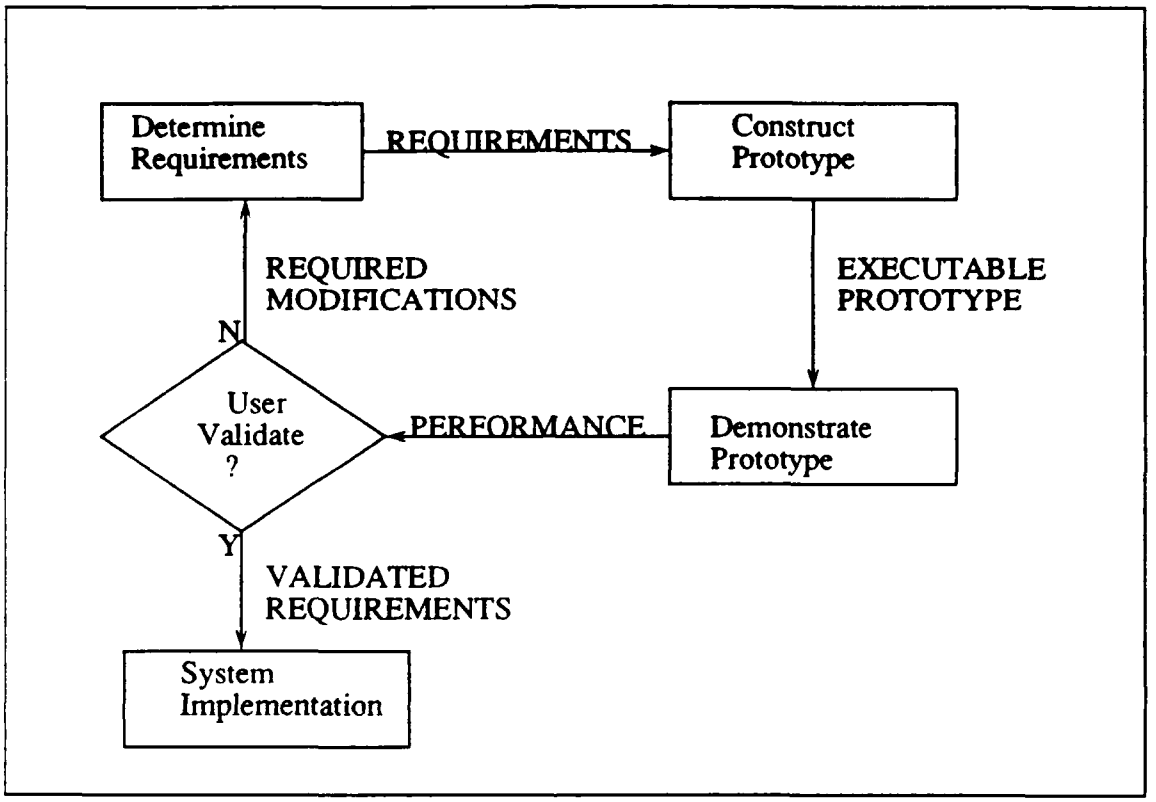


Figure 1. Rapid Prototyping Loop

The specification language (the Prototype System Description Language, PSDL) allows the designer to formally translate the user's requirements and concepts into specifications. The specifications can then be used to retrieve reusable software components from the software base. After all of the components of the prototype have been retrieved or constructed by the user, they are executed by the execution support system (ESS). The ESS is the focus of this thesis. It is composed of three systems, the translator, the static scheduler, and the dynamic scheduler. PSDL is an executable specification language whose execution is a combined effort of the translator, static scheduler, and dynamic scheduler. The translator translates the PSDL specifications into executable Ada modules while the static scheduler analyzes the PSDL

specifications and data-flow dependencies to construct a schedule of operation for the set of Ada modules with timing constraints. The dynamic scheduler schedules those Ada modules with no timing constraints. [Ref. 3:p. 71]

Successful execution of a CAPS prototype is dependent on two phases, translation and scheduling. The subject of this thesis is the construction of a translator to successfully perform the translation phase. The translation process begins with the input of a program written in PSDL. The program is parsed into its syntactic constructs, then those constructs are analyzed and Ada code is produced to represent them. Translation also means recognizing built-in PSDL constructs and using reusable generic components to implement the constructs. The objective of this thesis is the partial implementation of a translator for the CAPS system.

An overview of the CAPS system components, the Kodiyak application generator, and PSDL are presented in Chapter II. PSDL constructs and their semantic descriptions are presented in Chapter III. The implementations of PSDL constructs are presented in Chapter IV. The implementation of the translation process will be presented in Chapter V. Conclusions are presented in Chapter VI.

II. BACKGROUND

A. CAPS OVERVIEW

The computer aided prototyping system (CAPS) is a rapid prototyping tool presently being developed. The objective of CAPS is to reduce a software designer's efforts by automating time consuming tasks in conventional prototyping, such as turning specifications into prototypes, modifying prototypes, and searching for available reusable components. To support this prototyping method, several tools are included as subsystems of CAPS which are integrated together to form the functional group called CAPS. The components of CAPS include

- A specification language.
- A user interface to speed up design entry and prevent syntax errors.
- An execution support system to demonstrate and measure prototype behavior and to perform static analysis of the prototype design.
- A design management system to manage reusable software components and design data.
- A software base to store reusable components.
- A design data base to store the prototype design. [Ref. 3: p. 67-68]

Figure 2 shows the CAPS system architecture. Following is an overview of the functions of each of the components of the CAPS architecture

The user interface consists of a syntax-directed editor [Ref. 4] and a graphical editor [Ref. 5]. These two tools are used together to generate a design which is free of syntax errors as quickly as possible. The graphical editor allows the designer to enter his design as a data flow diagram so that relationships between PSDL components can be visualized by the designer and by the user. The syntax-directed editor speeds up the process of entering PSDL textual specifications. [Ref. 6:p. 6-7]

The rewrite subsystem translates the specifications produced in the user interface into normalized specifications which can be used by the design management system to retrieve components from the software base. Using normalized

specifications results in fewer keys to use for the search operation on the data base. [Ref. 3:p. 69]

The design management system [Ref. 7] is an object-oriented database management system. It is responsible for organizing, retrieving, and instantiating reusable components from the software base as well as managing versions, refinements, and alternatives of prototypes in the design database. It provides special purpose operations to compose components, browse the software base, and manipulate the normalized specifications. [Ref. 3:p. 69]

The software database is an object-oriented software base containing reusable components. The software base provides the ability to browse, select, and retrieve components efficiently. The database is searched by providing a normalized specification which can be mapped to one or more software implementations. The entire set of retrieved reusable components are then presented to the designer so that he may select a choice from the set. [Ref. 3:p. 70-71]

The execution support system provides the means to demonstrate the actual performance of the specifications produced in the user interface. There are three components of the execution support system, the translator, the static scheduler, and the dynamic scheduler.

The translator generates Ada code to bind together the reusable components extracted from the software database. Its main functions are to implement data streams and control constraints [Ref. 3:p. 71]. The output of the translator is a set of Ada packages and procedures that can be activated in a specific order to effect execution of the prototype.

The static scheduler [Ref. 8] analyzes the data flow structure and control flow structure of the input PSDL program and determines an order of execution, or schedule, for the PSDL modules which meets the data flow and timing constraints of the prototype.

The dynamic scheduler [Ref. 9] schedules those PSDL modules that have no timing constraints. It recognizes when a time slot is available during prototype execution and schedules non-time critical modules in those slots.

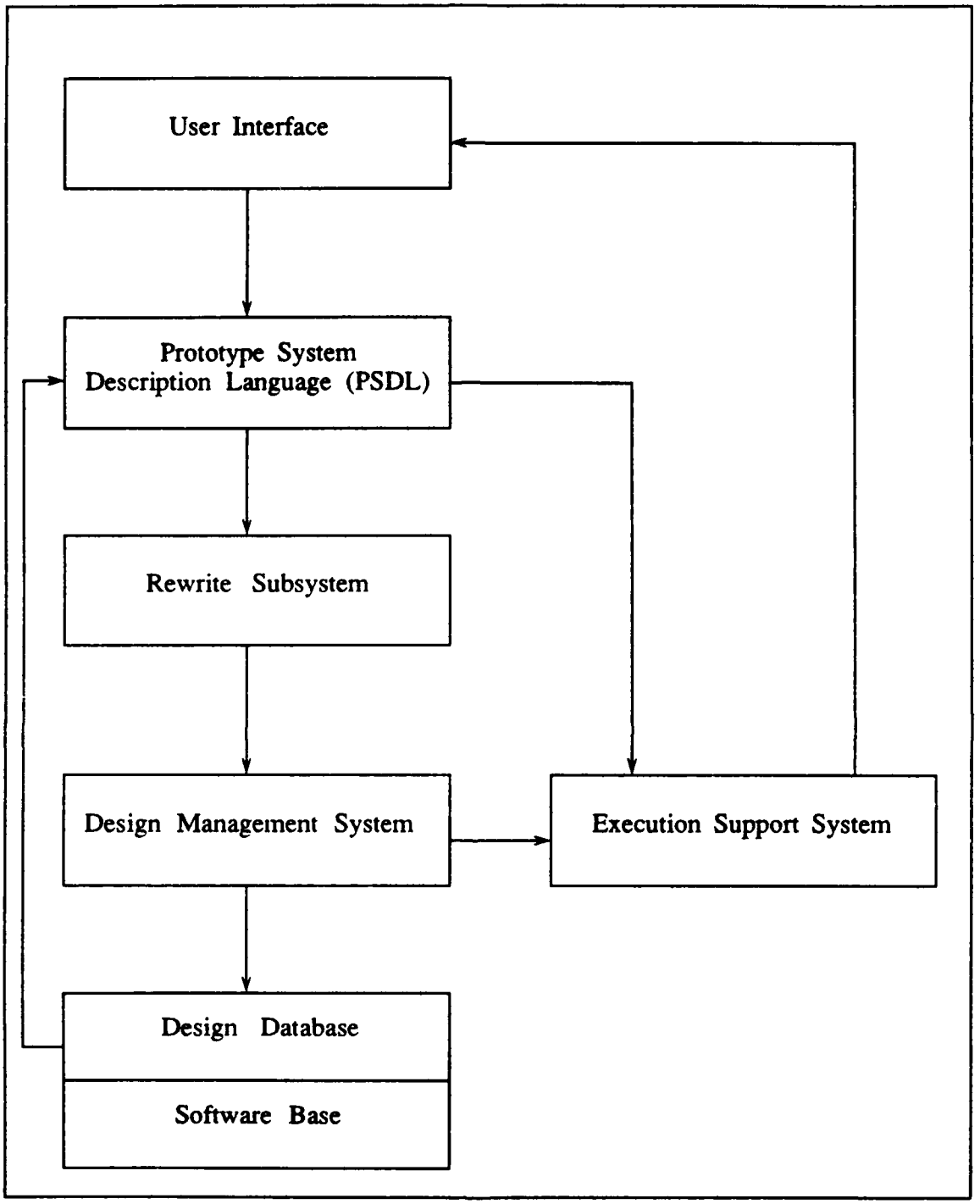


Figure 2. CAPS System Architecture

B. PSDL

The translator's purpose is to translate a PSDL specification into a set of Ada procedures which, when executed in a specific order, will effectively execute the prototype. The translation process is completely dependent on the semantics of PSDL.

There are several constructs in PSDL that a designer uses to build a prototype specification. These constructs include data streams, timers, exceptions, and operators. The translator actually builds the operator construct itself during translation, but before it can do this, each of the other PSDL constructs must be previously constructed and available. Construction of a PSDL operator involves instantiating various Ada representations of PSDL constructs and manipulating those constructs to provide a supporting framework in which a PSDL operator can be simulated.

1. PSDL Conceptual Model

PSDL is based on a conceptual model existing as a network of operators and data streams. Operators in the network communicate with each other through data streams [Ref. 6:p. 11]. An operator is a function which performs a specific user defined task. It requires a set of input values and produces a set of output values each of which may be empty. A data stream is a conduit for moving data values between operators. They are the source of an operator's inputs and they are the sink into which an operator places its output.

PSDL can be represented as a computational model described by an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is a set of vertices, E is a set of edges, T(v) is the maximum execution time for each vertex v, and C(v) is a set of control constraints for each vertex v. In this graph a vertex represents an operator and an edge is a data stream.

Figure 3 is an example of a PSDL graph with operators, A, B, and C, and data streams, a, b, c, d, e, and f, connecting the operators. The graph also indicates timing constraints, 10 ms for A and C, 20 ms for B. Control constraints are provided for A and B, also. The operator A receives its input data on data stream a, processes the data and outputs two results on data streams c and d. This entire process must occur in ten milliseconds or less according to the associated timing constraint.

Normally, control constraints are not described on the graph because they can be numerous and are described textually within a PSDL specification. A graph without control constraints is called an enhanced data flow diagram. [Ref. 3:p. 69]

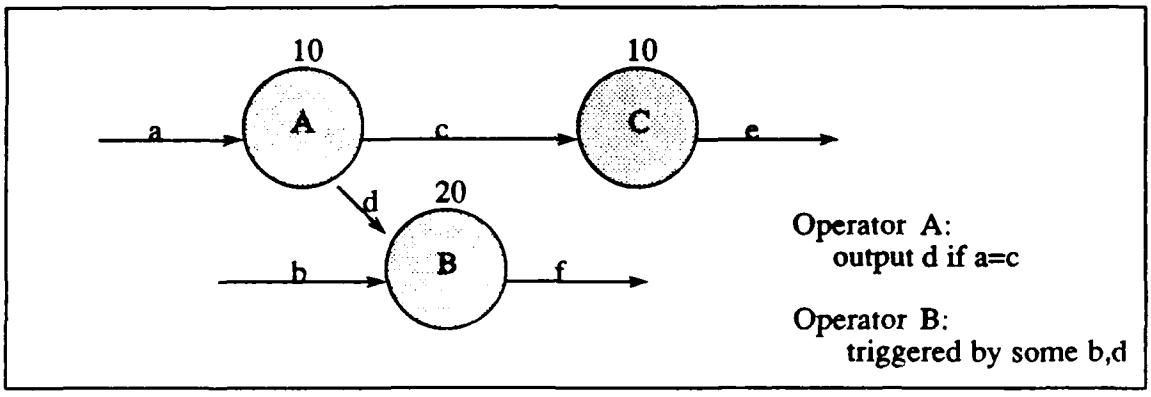


Figure 3. Enhanced Data Flow Diagram With Control Constraints

2. Operators

Operators in PSDL are either functions or state machines. A function produces output whose value is dependent solely on the input values. A state machine produces output whose value depends on the input data values and on internal state values. Operators can be triggered by the arrival of input data values on their data streams or by a periodic timing constraint which says the operator must fire at a regular time interval.

PSDL operators may also be atomic or composite. Atomic operators represent a single operation and cannot be decomposed into subcomponents. A composite operator represents the network of subcomponents and data streams into which it can be decomposed. [Ref. 6:p. 11-12]

3. Data Streams

Data streams represent sequential data flow mechanisms which move data between operators. There are two basic types of data streams, sampled and data flow. Data flow data streams are similar to first-in-first-out (FIFO) queues with a length of one. Any value placed into the queue must be read by another operator before another data value is placed into the queue again. Sampled data streams may be

considered as a single cell which may be read from or written to at any time and as often as desired. [Ref. 6:p. 12-13]

4. Timing Constraints

Timing constraints are an essential part of specifying real-time systems. They impose an order on operator firing which is based on timing or control flow rather than data flow. The three basic types of timing constraints are MAXIMUM EXECUTION TIME, MAXIMUM RESPONSE TIME, and MINIMUM CALLING PERIOD. The maximum execution time is an upper bound on the length of time from the instant when an operator begins execution to the instant it completes. [Ref. 6:p. 20]

The maximum response time has two interpretations depending on the type of operator it applies to. For sporadic operators which have no period, it is an upper bound on the length of time between the arrival of one or more new data values on an input data stream and the time when the final output value is placed on an output data stream in response to the new input values. For periodic operators, maximum response time is an upper bound on the time between the beginning of a period and the time when the last output value is placed onto a data stream of that operator during that period. [Ref. 6:p. 20]

The minimum calling period is a lower bound for sporadic operators on the time between the arrival of one set of inputs and the arrival of the next set. [Ref. 6:p. 20]

5. Control Constraints

Control constraints are the mechanisms which refine and adapt the behavior of PSDL operators. They specify how an operator may be fired (by data flow or control flow), how exceptions may be raised, and how or when data may be placed onto output data streams.

With the exception of timing constraints, all of the features of the PSDL computational model will be discussed in detail in Chapters III and IV. For a more detailed discussion on timing constraints see [Ref. 8] and [Ref. 10]

C. KODIYAK TRANSLATOR GENERATOR

Kodiyak is a fourth generation language designed for producing language translators and prototyping languages [Ref. 11:p. 1]. It was developed at the University of Minnesota and is based on Knuth's description of attribute grammars [Ref. 12].

Very generally speaking, Knuth wrote that a context-free language can be represented as a set of grammar rules and that by systematically applying those grammar rules to an input string, the input string can be uniquely represented as an abstract syntax tree [Ref. 12:p. 127-128]. Knuth goes on to say that attributes can be assigned to the nodes of the abstract syntax tree and that the values of the attributes can be determined in two ways.

The first method is by synthesized attributes which assigns values to the attributes of a node based on the attribute values of the node's descendants. The second method is by inherited attributes where a node's attribute values are based on the attribute values of its parent node. Synthesized attributes are evaluated from the bottom up in the abstract syntax tree while inherited attributes are evaluated from the top down. [Ref. 12:p. 130]

The point of Knuth's paper is that attributes can be assigned to the nodes of an abstract syntax tree and those attributes can be defined in terms of other attributes in the tree. Thus, the root of the tree can be given an attribute whose value is based on the collective values of all the nodes in the tree. That attribute of the root node can be used to assign a semantic meaning to the tree [Ref. 12:p. 132]

Kodiyak puts this theory into practice in three phases. First, it performs a lexical analysis on the input string converting it to tokens; second, it parses the tokens into an abstract syntax tree; and third, it proceeds to move about the tree evaluating attributes at each node until all attributes have been evaluated. The meaning of the input string can then be represented as the collective value of the attributes at the root node.

Kodiyak is a UNIX-based tool which is built on top of two other tools, LEX (a lexical analyzer tool) and YACC (a parser generator tool). Every Kodiyak program has three sections. The first section describes the features of the lexical scanner which is used to translate the input source text into tokens. The second section describes the

attributes assigned to each grammatical part in the language. The third section describes the grammar and attribute equations which are used to determine values of attributes throughout the tree. [Ref. 11:p. 1]

In the lexical scanner section of a Kodiyak program, rules are given which define the token terminal symbols of the source language and their token representations. Terminal symbols are defined to be regular expressions and are found in the leaf nodes of the abstract syntax tree. Token definitions take the form shown in Figure 4 where TERMINAL_NAME is the name of the token and REGULAR_EXPRESSION is the definition of the token. For example, two lexical definitions used in this thesis are

```
OPERATOR    "operator" | "OPERATOR"
SOME        "by some" | "BY SOME".
```

The first definition defines a token named OPERATOR which is created anytime the regular expression "operator" or "OPERATOR" is found in the input text. The second definition defines a token named SOME which is defined anytime the regular expression "by some" or "BY SOME" is found in the input text. It is by definitions such as these, that Kodiyak reads an input text file and converts it to a stream of tokens. [Ref. 11:p. 2-6]

The attribute declaration section lists all of the non-terminal symbols in the input grammar along with the attributes associated with each symbol. The non-terminal symbols are the interior nodes of an abstract syntax tree and are normally found on the left side of a typical grammar rule. Kodiyak also allows terminal symbols to have attributes if desired. The statements

```
psdl {trn : string;};
ID   { %text : string;
       %line : int;
       value : int;};
```

are two examples of attribute declarations. The first statement declares a non-terminal symbol named **psdl** which has only one attribute named **trn** of type string. Kodiyak only supports two primitive types, string and integer. The second example declares the attributes for a terminal symbol. An ID token has three attributes,

%text which is type string, **%line** which is type integer, and **value** which is type integer. The **%line** and **%text** attributes are predefined in Kodiyak.

```
TERMINAL_NAME        : REGULAR_EXPRESSION
```

Figure 4. Kodiyak Lexical Definition Form

The **%line** attribute always contains the line number in the input text where the associated token was found. The **%text** attribute always contains the actual text which matched the regular expression definition for the token. [Ref. 11:p. 2-6]

The final section of a Kodiyak program is the attribute grammar section. This section defines the syntax and semantics of the translation. It consists of a set of grammar productions in a form similar to BNF and sets of equations defining attributes [Ref. 11:p. 8]. The example in Figure 5 is an actual rule used in the translator in this thesis. It is based on the PSDL grammar rule:

timer_op ::= READ | RESET | START | STOP

where timer_op is a non-terminal and READ, RESET, START, and STOP are all terminal tokens. The rule states that a timer_op is recognized if any one of the tokens, READ, RESET, START or STOP is found in the token stream. The associated attribute equations are enclosed in curly braces for each possible rule.

```
timer_op: READ
            { timer_op.tm = "PSDL_TIMER.READ"; }
        | RESET
            { timer_op.tm = "PSDL_TIMER.RESET"; }
        | START
            { timer_op.tm = "PSDL_TIMER.START"; }
        | STOP
            { timer_op.tm = "PSDL_TIMER.STOP"; }
        ;
```

Figure 5. Kodiyak Attribute Equation Example

The example shows that if timer_op produces a READ token, then the **trn** attribute of timer_op will be assigned the string value of "PSDL_TIMER.READ".

Without Kodiyak, the implementation of a translator for PSDL would be a much more formidable task than it is. Essentially, a lexical analyzer and parser would have to be written for PSDL, which are non-trivial tasks, then a tool would be built which can traverse an abstract syntax tree and generate code. This would effectively be equivalent to writing a compiler for PSDL. Kodiyak allows a programmer to concentrate on the translation process, i.e., attribute evaluation while it handles the compiler-like functions.

III. MAJOR PSDL CONSTRUCTS

The constructs used in PSDL include operators, data streams, state variables, timers, and exceptions. The execution of a PSDL prototype is actually just the integration and manipulation of these constructs according to a set of control constraints. With the exception of operators, each of these constructs is represented by an abstract data type in Ada. The following is an informal discussion of the semantics of the major PSDL constructs.

A. DATA STREAM ABSTRACTIONS IN PSDL

The edges of the PSDL computational model represent data streams. Data streams are the mechanism by which two or more operators can communicate explicitly. Data streams can carry individual data values of abstract data types, values of built-in PSDL type EXCEPTION, and values of the built-in data types of Ada. Data streams can be described as a pipeline connecting exactly one producer with exactly one consumer [Ref 6:p. 12].

Figure 6 is an example of a producer operator, op1, connected to a consumer operator, op2, by the data stream D. The producer operator has produced the values x1 and x2 in order. The pipeline property of data stream D dictates that value x1 must arrive at the consumer operator before value x2 because x1 was created first. Another important property of data streams is that they may carry no more than one data value at a time.

When an operator produces a data value, it writes that value onto a data stream. When an operator consumes a data value, it reads that value from a data stream [Ref. 13].

As implied in the previous paragraph, data streams have operations for reading them and writing to them. There is also a third operation for checking them to verify if there is a new, or fresh, data value currently on the data stream.

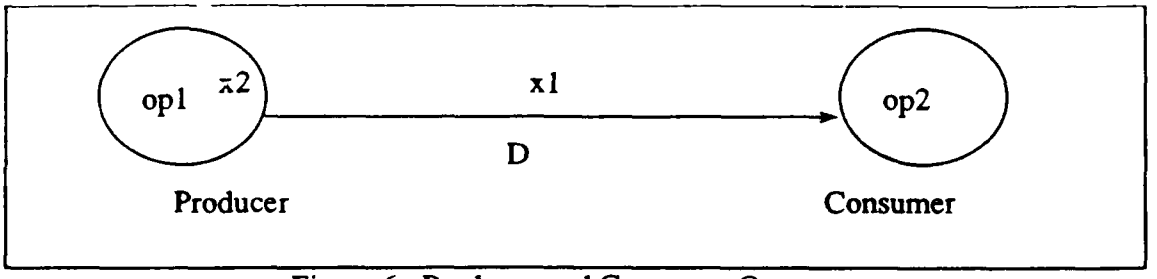


Figure 6. Producer and Consumer Operators

There are two types of errors associated with data streams, BUFFER_OVERFLOW, and BUFFER_UNDERFLOW. BUFFER_OVERFLOW occurs when an operator attempts to write to a data stream that currently has a value on it.

BUFFER_UNDERFLOW is caused when an operator attempts to read a value from an empty data stream. Applying the check operation to a data stream will never cause either of these two errors because it neither reads nor writes to the data stream. [Ref. 13]

There are two basic kinds of data streams, SAMPLED STREAM and DATA FLOW STREAM. A sampled data stream has a continuous nature in that once a data value has been written onto it, that data value will remain on the data stream until it is overwritten with another value. Even after a value is read from a sampled data stream, a copy of that value remains on the data stream and can be read as often as needed. [Ref. 13]

Since data values remain on a sampled data stream after they have been consumed, it is not possible to have a BUFFER_UNDERFLOW error on a sampled data stream after it has been initialized once. In an uninitialized state, sampled data streams do not have values on them and therefore the BUFFER_UNDERFLOW error could occur if a read operation is performed on the data stream before the first write operation. The BUFFER_OVERFLOW error cannot occur on a sampled data stream at all because the write operation is always legal on a sampled stream.

Data flow streams have a more discrete nature about them because they are a first-in-first-out data structure. A data value can be written onto a data flow stream only if the stream is empty, and conversely, a data value can be read from a data flow

stream only if a data value presently exists on the stream. After a data value is read from a data flow stream, the stream is left empty. [Ref. 13]

BUFFER_UNDERFLOW and BUFFER_OVERFLOW can be more easily generated in data flow streams than in sampled streams because of their first-in-first-out nature. Any attempt to read from an empty data flow stream will cause BUFFER_UNDERFLOW. Any attempt to write to a data flow stream which currently has a value on it will cause BUFFER_OVERFLOW. Because of the ease of generating these errors in data flow streams, care must be exercised when using them in a prototype. The timing of the producer and consumer operators must be synchronized so that the rate of data consumption on a data flow stream is always equal to the rate of data production. [Ref. 6:p. 13]

Whether a data stream is a sampled or a data flow stream is determined implicitly by the context in which the stream is used in a PSDL program. Specifically, if a stream is used in an operator triggering condition that specifies the operator is triggered **BY ALL**, then the stream is classified as a data flow stream. (Operators and triggering conditions are described in a later section.) All other data streams not so used are sampled data streams by default.

There are four places to declare data streams in a PSDL operator:

- In an INPUT statement.
- In an OUTPUT statement.
- In a STATES statement.
- In a DATA STREAM statement.

Data stream visibility within an operator is somewhat hierarchical yet it differs from the hierarchical visibility rules that may apply in structured languages like Pascal or Ada. In PSDL, data streams which are visible in any operator are also visible to that operator's immediate subcomponents but not necessarily to any of its second generation descendants. An operator which has a parent operator can selectively choose which data streams in its parent to make visible within itself, thus, if it ignores any data stream in its parent, that data stream will not be visible to any of the current operator's descendants.

INPUT and OUTPUT statements in a PSDL operator specification declare data streams used by an operator which are external to that operator. That is, these are the data streams belonging to the parent operator which are selected to be visible in the child operator. The direction of data flow is dictated by the INPUT and OUTPUT statements. An INPUT statement declares that data can only be read from its data streams. Likewise, an OUTPUT statement declares that data can only be written to its data streams. The forms of the INPUT and OUTPUT statements are shown in Figure 7.

The STATES and DATA STREAM statements are also shown in Figure 7. These statements declare data streams that are local to the current operator. The data streams declared in the STATES statement are like any other data stream except that they are initialized automatically. They are used to carry values of state variables which are described later in the discussion of PSDL operators. The DATA STREAM statement declares all local data streams that do not carry state variables.

```
INPUT  id_list : type_name [, id_list : type_name]*
OUTPUT id_list : type_name [, id_list : type_name]*
STATES id_list : type_name INITIALLY initial_values
DATA STREAM id_list : type_name [, id_list : type_name]*
```

Figure 7. Method of Declaring Data Streams

B. TIMERS

In PSDL, a timer is a built-in data type. The timer data type acts as a simple digital stopwatch to record elapsed times. The operations available on timers include:

- RESET
- START
- STOP
- READ

Semantically, a timer can be described as a state machine (see Figure 8) having three states, an initial state, a running state, and a stopped state.

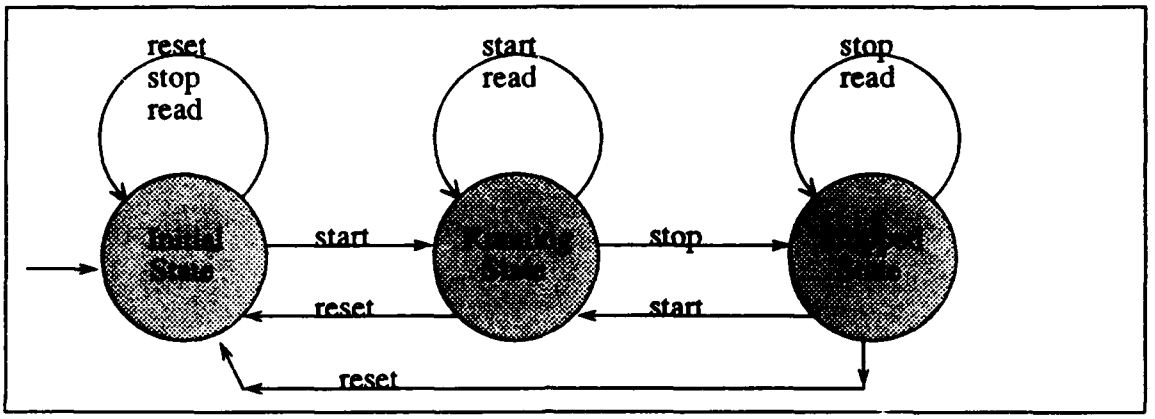


Figure 8. Timer State Machine

In the initial state the timer has a value of zero. All timers are initialized to begin in the initial state and thereafter will only return to the initial state when the reset operation is performed.

The timer transitions to the running state whenever the start operation is performed. The value of the timer in the running state constantly increases with time, but its instantaneous value can be obtained by the read operation.

The stopped state can only be reached from the running state and is done so when a stop operation is performed. In this state, the timer is effectively idle and not measuring elapsed time, that is, its value is constant.

The value of a timer can always be obtained by the read operation. Note that the read operation does not cause a state transition regardless of the state in which it is performed. The value of a timer is always the amount of time the timer has spent in the running state since the last transition out of the initial state.

Timers are declared inside a PSDL operator by the TIMER statement whose form is shown in Figure 9. It is a simple statement that declares any number of PSDL timers inside of an operator. Unlike data streams, timers fully utilize hierarchical visibility rules. A timer is visible in the operator in which it is declared and all of its subcomponents on all levels [Ref. 6:p. 18]. Presently, an implementation restriction in this thesis dictates that timer names cannot be duplicated in a PSDL program.

All references to a specific timer name throughout a PSDL program are references to the same timer.

```
TIMER   t1, t2,...,tN
```

Figure 9. Timer Declaration

Timer values can be passed along data streams so that a timer's value can be made visible outside of its scope. The timer itself, however, cannot be transmitted because it is a state machine, not a value.

C. PSDL EXCEPTIONS

PSDL exceptions as described by Luqi in [Ref. 6] are special data types that may be written to any data stream regardless of the stream's normal data type. When a PSDL exception is raised in an atomic operator it is immediately output onto all of the operator's output data streams. This poses a problem in designing downstream operators in the data flow. Any time an operator raises a PSDL exception, all operators immediately downstream will have a PSDL exception on one or more of their input streams. The designer must consider such a case when writing PSDL operators. A PSDL system is a network of atomic operators connected by data streams. Actual systems could have very complicated networks. Ensuring that the possibility of a PSDL exception value is handled everywhere in the system could be a formidable task.

This thesis describes an alternative method for handling PSDL exceptions. The principle difference between this method and Luqi's is that PSDL exceptions are no longer allowed to share data streams with normal data values. An operator must explicitly declare any external or local data steams which carry the PSDL_EXCEPTION data type. The alternate design is intended to modify the original design of PSDL exceptions as little as possible, given the difference stated earlier.

Requiring operators to transmit PSDL exceptions only on data streams that are declared for the PSDL_EXCEPTION data type means that exception communications are now explicit. If an operator is allowed to raise a PSDL exception it can only

communicate that exception to operators which are directly connected to it via an exception data stream. A PSDL exception handler, therefore, is any PSDL operator which has an input exception stream declared in the INPUT statement in its PSDL specification. The remainder of this section presents PSDL exceptions in terms of the alternative method.

PSDL exceptions are values of a built-in data type called PSDL_EXCEPTION. They are used to denote the occurrence of a value or situation that the designer does not want to occur during execution of the prototype. For example, the value on a data stream may be out of limits or a data stream may not have a value when it is expected to have one. The PSDL_EXCEPTION data type has operations for creating an exception, raising an exception, and detecting that an exception has been raised. A PSDL exception has two states, raised and cleared. A raised PSDL exception is one which is active in the same manner that an Ada exception is raised. Conversely, a cleared PSDL exception is one that is not raised.

PSDL exceptions can be raised two different ways. The first is by the EXCEPTION statement in an operator's control constraints. The second method of raising a PSDL exception is through an implicit conversion of an Ada exception to a PSDL exception. Atomic operators may raise exceptions in the underlying language [Ref. 6:p. 19] and the designer has the option of handling the exception in the underlying language or in PSDL.

When a PSDL exception is raised it is written to one or more data streams. The operator which was executing at the time of the exception will be terminated after the exceptions have been transmitted. When a PSDL exception is raised in an EXCEPTION statement, the designer has control over which streams the exception is transmitted. He may selectively choose one or more exception data streams to write the exception to. PSDL exceptions raised as a result of an Ada exception are not subject to such control. Instead, they are transmitted on all output data streams which are declared to be of the PSDL_EXCEPTION data type. If no output exception data streams are available in the operator, there is no way of communicating the converted exception. In that case, the Ada exception is re-raised and handled elsewhere in the execution support system.

Exception handling in PSDL is not immediate as in Ada. CAPS places a constraint on prototype execution that adversely affects execution of exception handlers. Consider the operation of exception handlers in Ada. When an Ada exception is raised, normal execution control flow is interrupted so that an exception handler can be located for the raised exception and executed. It is both desirable and good language design, that exceptions be handled immediately rather than allowed to linger about in a program. In CAPS, however, execution is bound to a scheduled sequence produced by the static scheduler and the dynamic scheduler [Ref. 8] which cannot be altered. PSDL exception handlers are not given any special consideration by the static scheduler; they are simply operators that are scheduled to execute in a predefined order, they cannot be invoked immediately as they are needed. Thus, PSDL exceptions are allowed to be raised but their handling is subject to the static and dynamic schedules of execution.

Detecting whether a PSDL exception has been raised is a special operation that is performed during evaluation of boolean expressions. It is a construct consisting of the data stream name containing the exception, followed by a colon, followed by the name of the exception. For example,

OPERATOR a TRIGGERED IF In_Exception : Out_of_Range

demonstrates a boolean expression which checks to see if the PSDL exception, **Out_of_Range**, is raised on the data stream called **In_Exception**. The data stream, **In_Exception** must have been declared as an input data stream of type PSDL_EXCEPTION in this example.

A PSDL exception is effectively cleared when it is read from a data stream. A PSDL exception is read anytime its value is used in a boolean expression other than an operator data trigger.

PSDL exception visibility follows the same rules as data streams. Any exception written to a target data stream is visible only to the operators having access to the target data stream.

It is required to include an exception handler in any decomposition which is capable of producing a PSDL exception. Figure 10 shows an example of this technique. The operator, A, decomposes into three operators, B, C, and D. Operator B is

capable of raising a PSDL exception, thus it has an exception data stream named **error** connecting it to the operator EH. EH is an exception handler which is triggered by the presence of a value on the incoming data stream. The static scheduler will schedule EH to execute sometime after operator B, thus handling the error as soon as possible and minimizing the time it is allowed to linger in the system.

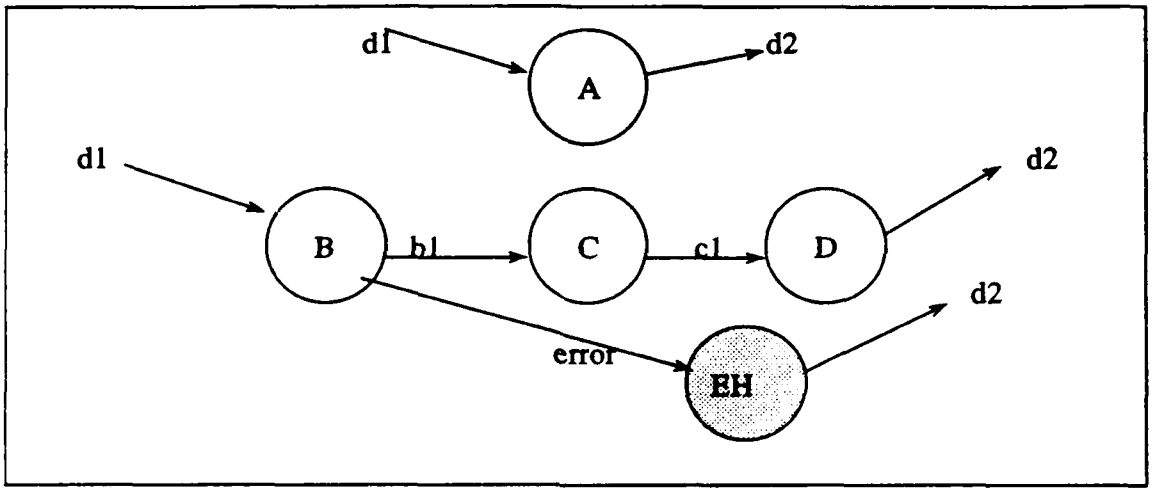


Figure 10. Local Exception Handler

D. OPERATOR ABSTRACTIONS IN PSDL

The vertices of the PSDL computational model represent operators. Until now operators have been only referred to as producers or consumers, however, there is much more to an operator than the fact that it produces and/or consumes data. Operators are different from the other constructs in that they are not a PSDL type; instead, operators are objects which manipulate the other PSDL constructs in unique ways. How an operator manipulates PSDL constructs like data streams, timers, and exceptions partially defines the behavior of the operator.

1. Operator Classification

PSDL operators may be classified by three criteria:

- Function or state machine operator.
- Atomic or composite operator.
- Periodic or sporadic operator.

a. Functions and State Machines

Classifying an operator as a function or a state machine depends on whether the operator has an internal state. A function operator does not have an internal state. It simply consumes its input and produces an output; the value of the output depends solely on the value of the input. Specifically, a function will produce a one-to-one mapping from any set of input data values, x1,x2,...,xN, to a set of output values, y1,y2,...,yM. A state machine, on the other hand, has an internal state which affects the values of its output. A state machine will produce a one-to-many mapping from a set of input data values to a set of output values. The value of the output data is a function of the input data and the internal state. [Ref. 6:p. 15]

The internal state in a PSDL state machine operator is represented by a STATE VARIABLE. Recall that a state variable is a data stream which is automatically initialized. State variables are declared in a PSDL operator by the STATES statement as shown previously in Figure 7. The STATES statement declares a list of data streams, which will be local to the current operator, and a list of initial values to be placed into the data streams before the prototype begins execution. A state machine can be graphically represented as an enhanced data flow diagram with a self-loop as in Figure 11. The self-loop sv1 is the data stream carrying the state variable.

b. Atomic and Composite Operators

This classification depends on how an operator decomposes. Atomic operators cannot be decomposed into a lower level of functionality while composite operators can be decomposed into a set of lower level operators whose execution as a whole simulates execution of the composite operator. Figure 12 shows an example of a composite operator, A, and its corresponding decomposition into two lower level operators, A1 and A2. Operator A1 is also a composite operator and so decomposes

into operators A11 and A12. Operator A2 is an atomic operator and cannot be decomposed into any lower level of detail. A PSDL program is nothing more than a hierarchical decomposition of a single composite operator into a network of atomic operators. The ordered execution of the atomic operators effectively simulates execution of the highest level operator. [Ref. 6]

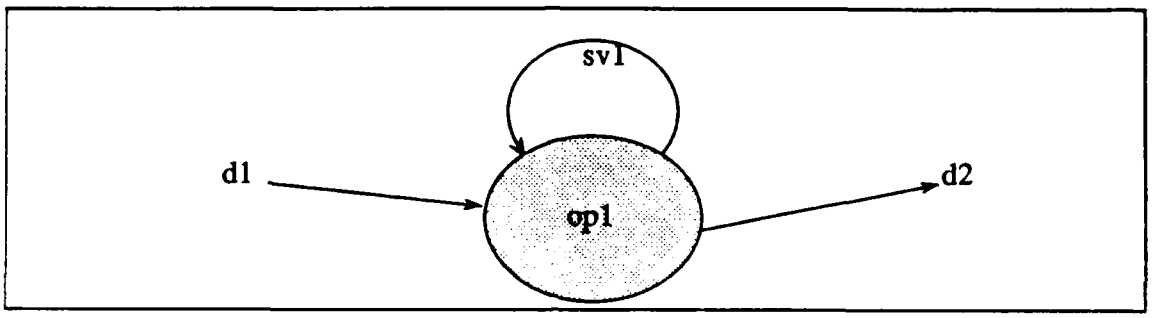


Figure 11. Operator With State Variable

c. Periodic and Sporadic Operators

This classifies an operator by its timing constraints. A periodic operator has timing requirements which state that the operator must be invoked regularly within a certain period and that it must complete execution within that period. Sporadic operators have no such periodic constraints; instead, they are required to execute when there are fresh data on the input data streams. The translator is concerned with an operator's behavior as it executes, not how often it executes or how long it executes, therefore, this classification is irrelevant to the translation process. For a more detailed discussion concerning timing constraints and their implementation refer to [Ref. 8].

2. Operator Control Constraints

To support real-time systems modeling in PSDL, a powerful set of control constraints are available which impart a specific behavior upon atomic operators. Through control constraints several aspects of an operator's behavior can be specified, such as the classification of the operator, the operator's triggering conditions, and the operator's output conditions [Ref. 13]. The translator is concerned with recognizing control constraints that have been placed on an operator, which are not

timing related, and then generating the Ada code required to simulate the desired operator behavior. There are four types of control constraints to be specified:

- Operator triggering conditions.
- Exception conditions.
- Timer conditions.
- Operator output conditions.

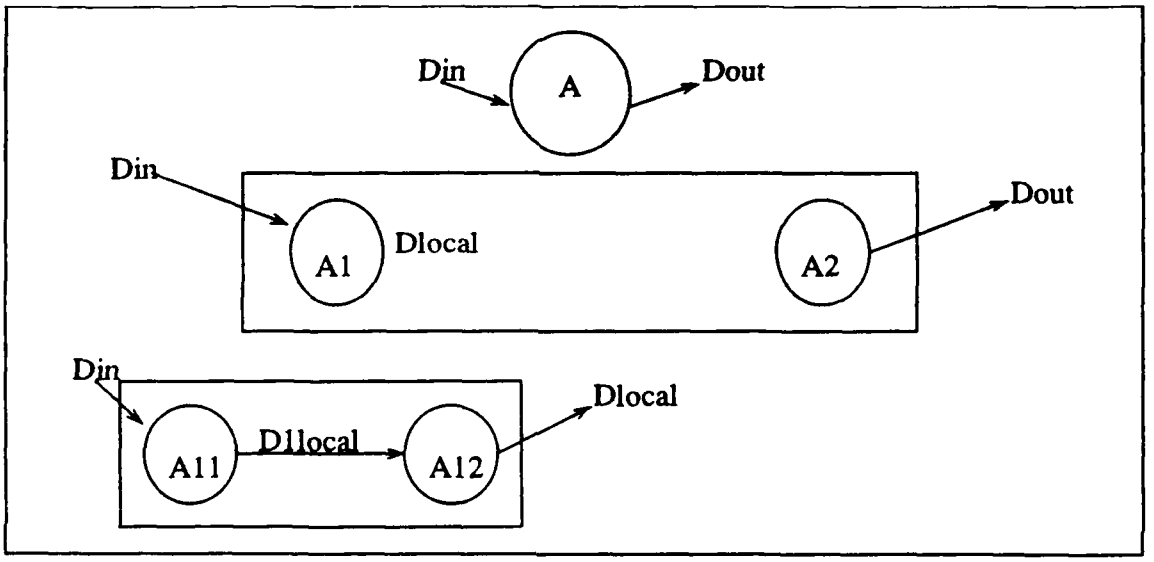


Figure 12. Operator Decomposition

a. Operator Triggering Conditions

A triggering condition has two parts, a data trigger and an input guard [Ref. 14]. The data trigger is designed to control the initiation of execution of an operator based on the presence of new input data values. A new data value on a data stream is any data value that has not been read at least once. The input guard is a secondary condition which makes an operator's execution dependent on the values of its input data. The triggering condition is a means of specifying data flow control among operators. The form of a PSDL triggering condition is shown in Figure 13.

```
Triggering_Condition =
    OPERATOR Atomic_Name TRIGGERED Data_Trigger IF Input_Guard
```

Figure 13. PSDL Triggering Condition

The data trigger part of a triggering condition can be further broken down into a **BY ALL** or a **BY SOME** condition. Referring to the operator model in Figure 14, the PSDL statement

OPERATOR op1 TRIGGERED BY ALL A,B

establishes a data trigger on operator op1 that allows it to execute only if there are new data values on both of the input data streams A and B. The contents of data stream C are not considered in the trigger condition. If either A or B does not have a fresh data value then the trigger condition is not satisfied and the operator will not be allowed to execute at all. The PSDL statement

OPERATOR op1 TRIGGERED BY SOME A,B

establishes a data trigger on operator op1 that allows it to execute only if there is new data on either data stream A or B. Again, the contents of data stream C are not considered in the trigger condition and have no effect on it. Conversely, if neither data stream A nor B has a fresh data value on it the trigger condition is not satisfied and the operator will not execute.

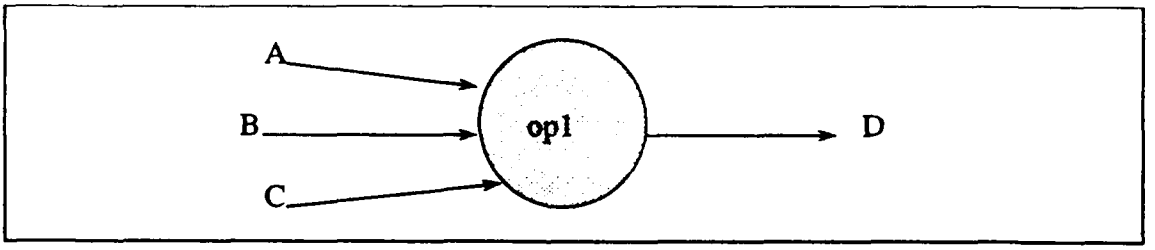


Figure 14. PSDL Operator Model

When a data trigger is satisfied, the operator will initiate execution by consuming the values on all of its input data streams. An unsatisfied data trigger results in an operator not executing at all and its data streams are left intact. The data trigger is an optional feature of the operator triggering condition. A null data trigger

effectively always evaluates to true and the operator is allowed to initiate execution regardless of the state of its data streams. Care should be taken in such cases not to consume a value from an empty data stream. Doing so results in a buffer error and the prototype's execution will be interrupted.

Assuming that an operator's data trigger condition is satisfied, the operator will consume its input data values and proceed to evaluate the input guard of the triggering condition. The input guard is a boolean expression dependent only upon the input values [Ref. 14]. Referring again to Figure 14, consider the PSDL statement

OPERATOR op1 TRIGGERED BY ALL A,B,C IF (A > B) AND(C > 0)

The operator op1 will first read all of its input data streams A, B, and C only if they all contain fresh data values. Assuming that the data trigger is satisfied, the operator must then evaluate the input guard, (A > B) AND (C > 0). If the values of A, B, and C satisfy this condition the operator is allowed to commit itself to complete execution and perform its function. If the input guard condition is not satisfied, the operator is not allowed to execute any farther and is terminated. The data streams will have been consumed and will not be restored to their original conditions prior to the trigger condition. The input guard is also optional. A null input guard effectively is always satisfied and the operator will be allowed to commit to complete execution. If both the data trigger and the input guard are null, the whole triggering condition is satisfied by default and the operator is allowed to execute unconditionally.

b. Exception Conditions

The form of the EXCEPTION statement is shown in Figure 15. It consists of an exception identifier, an optional target data stream, and an optional exception condition. An EXCEPTION statement may specify only a single exception name but several EXCEPTION statements may be included in a set of control constraints. The target data stream is an output data stream that has been declared to contain a PSDL_EXCEPTION type. The exception condition is a boolean expression dependent on values of input data streams, calculated output data streams, and timers.

```
Exception_Statement =
    EXCEPTION  Exception_Name ON Data_Stream IF Exception_Condition
```

Figure 15. PSDL EXCEPTION Statement

The example statement:

EXCEPTION Out_of_Range ON Excp IF (A<0) AND (A>10)

will raise a PSDL exception named **Out_of_Range** if the value of A is not in the range from zero to ten, inclusive. This example specifies a target data stream, named **Excp**, onto which the exception value is placed. This statement is used when the designer desires to explicitly transmit an exception to another operator which will act as an exception handler. The data stream, **Excp**, must have been declared as an output data stream or as a local data stream and it must be of type PSDL_EXCEPTION. Had no exception condition been specified, the PSDL exception, **Out_of_Range**, would have been raised unconditionally and output onto the target data stream, **Excp**.

It was stated earlier that an atomic operator may also generate exceptions in the underlying language. The designer has the option of handling those exceptions in the underlying language or he may convert them to PSDL exceptions. When an exception is produced in the underlying language (in this case, Ada) and is not handled in the atomic operator, an implicit control constraint is placed on the operator which forces the translator to build an exception closure which can trap the Ada exception and convert it into a PSDL exception. To do this, the designer must declare the Ada exception names in the EXCEPTIONS statement in a PSDL operator specification. Figure 14 shows the form of an EXCEPTIONS statement. It is a simple statement which lists one or more exception names which can be raised by an atomic operator. For example, a statement such as:

EXCEPTIONS Out_of_Range, No_Data

would declare two PSDL exceptions that could be raised by the atomic operator subcomponents of the operator containing the statement. When an atomic operator raises any of those exceptions, they will be trapped by the translator's code and a PSDL

exception with the same name will be transmitted on all output streams that are declared as PSDL_EXCEPTION streams.

```
Exception_Declaration =
   EXCEPTIONS e1, e2, ... ,eN
```

Figure 16. PSDL Exception Declaration

c. Timer Conditions

Timer conditions are control constraints that allow operators to manipulate any number of PSDL timers that are visible to them. The form of a timer condition is shown in Figure 17. It consists only of a PSDL timer function name followed by the target timer name and an optional timer guard. For example,

RESET TIMER t1 IF t1 > 45 sec

is a timer condition which results in the timer named t1 being reset when it reaches a value greater than 45 seconds. The timer guard in this example demonstrates a boolean operation on the value of a timer. If the condition is not satisfied, i.e., the value of t1 is less than or equal to 45 seconds, the timer operation is not executed. The timer, t1, must have been declared somewhere within the current operator's preceding hierarchical structure. Recall that timers are visible in the operator in which they are declared and all of that operator's subsequent descendants.

The timer guard is optional and a null timer guard is effectively equivalent to a true condition and the timer operation is unconditionally executed. Timer guards are boolean expressions which can be dependent on values of operator input data, output data, and visible timers.

```
Timer_Condition =
   Timer_Op TIMER  Timer_Name  IF Timer_Condition

Timer_Op =
   RESET | START | STOP | READ
```

Figure 17. PSDL Timer Condition Statement

d. Operator Output Conditions

An output condition takes the form shown in Figure 18 and consists of the PSDL keyword, OUTPUT, followed by an output data stream name, followed by an output guard. The output data stream name must be one of the data streams declared in the OUTPUT statement in the operator specification or it may be a local data stream declared in a DATA STREAM statement. Unlike timer guards, output guards are not optional. The output guard is a boolean expression which depends on the values of input data, output data, and timer values [Ref. 14].

```
Output_Condition =
    OUTPUT  Data_Stream  IF Output_Guard
```

Figure 18. PSDL Output Condition Statement

An output condition may be specified for any output data stream, but is not required. PSDL does not allow an operator to have more than one output condition per output data stream and it is illegal to have an output condition for an input data stream. If an output data stream does not have an output condition specified for it, a value will always be output to that stream unconditionally during operator execution.

Normally, output conditions are used when it is desired to restrict the data that is placed onto an output stream. To demonstrate the use of output conditions, consider the statement

OUTPUT D IF (D > 0) AND (A = B)

as applied to the operator in Figure 14. An output condition is established on operator op1's only output data stream, D. After the operator has executed its atomic module and calculated the value of D, it may write that value to its output data stream only if the output guard condition is satisfied, i.e., (D > 0) and (A = B). If this condition is not met, the operator will not write the value of D onto its data stream, otherwise it will. Had no output condition been specified for D, the operator would have written the calculated value of D onto its data stream unconditionally.

IV. IMPLEMENTATION OF PSDL CONSTRUCTS

A. TRANSLATOR INTERFACES

The translator is only a single component in the whole CAPS system which must be interfaced with other components of CAPS. Specifically, the translator must interface with the static scheduler, the dynamic scheduler, and the user interface system. This section describes the implicit and explicit interfaces with these components.

1. Translator/Static Scheduler Interface

The interface with the static scheduler is completely implicit. The static scheduler does not accept input from the translator and produce output based on it; instead, the static scheduler makes some assumptions about the structure and contents of the translator's output. The translator produces a set of Ada procedures which represent each unique atomic operator in a PSDL program. The static scheduler reads the same PSDL program and produces a set of procedure calls into the translator's output which will invoke the atomic procedures in an order sufficient to meet the data flow and timing requirements of the PSDL prototype. To make this relationship work three assumptions were agreed upon by the static scheduler designer and the translator designer:

- Atomic operator procedures adhere to a standard naming convention.
- Atomic operator procedure calls are parameterless.
- The translator output will have a standard name.

a. Atomic Operator Procedure Naming

When the translator produces Ada code for each atomic operator, it actually produces a harness of Ada code into which the actual atomic operator's code is placed. The harness consists of all the Ada code required to implement the control constraints on the atomic operators. The single-shot rule [Ref. 6:p. 25] states that no operator may be fired more than once inside any one parent composite operator; therefore it can be assured that all of the atomic operators in a single composite operator will be uniquely named. It is possible, however, that the same atomic

operator name may be used in other composite operators in a PSDL program. The set of control constraints may vary for each occurrence of that atomic operator. The translator would then produce several different procedures because, although the atomic code in each procedure is identical, the control constraints may vary and the surrounding harness code will be different in each instance. The problem to be solved in this case is how to make each of the atomic operators unique even when they use the same atomic name.

To ensure that an atomic operator procedure would be uniquely named even if the actual atomic code inside it has been duplicated elsewhere in the PSDL program, a standard naming convention was agreed upon. This standard specifies that an atomic operator name will be represented by the concatenation of its parent operator's name, followed by an underscore, followed by the atomic operator's name. An advantage of the naming schema chosen is that the names are meaningful to the user and can be used as is for debugging purposes.

An alternative solution to this problem is to build a tool which scans a PSDL program created in the user interface and physically maps each operator name to a unique name. The translator and static scheduler could then be guaranteed that no name in the entire PSDL input program would be duplicated. There would no longer be any need for a naming convention since all atomic operator names would be unique to begin with. Such a tool could easily be constructed using Kodiyak.

b. Procedure Parameter Passing

The second concern about the translator's output was whether or not the static scheduler should include parameters inside its atomic operator procedure calls. The translator was designed to account for parameters in atomic procedures so that the static scheduler could invoke these procedures without regard for their parameters. The static scheduler only needs to know the name of the atomic operator procedure and can invoke it with a parameterless procedure call.

Recall that the translator actually produces a harness for the atomic operator's Ada code. The translator can recognize an atomic operator's requirements for input and output parameters via the input and output data stream declarations in the atomic operator's PSDL specification. All this information is embedded into the

harness code as local information so the harness procedure itself requires no parameters. Thus when the static scheduler produces a call to an atomic operator using the naming convention described earlier, it is actually calling the harness procedure which in turn can call the actual atomic procedure and implement all control constraints associated with that operator.

c. Package Naming

After the translator has produced all of the atomic operator procedures required, it must package them in a manner that makes those procedure names visible to the static schedule. This is easily accomplished by placing the translator's output in an Ada package body. Ada requires that a package body be accompanied by a package specification as well. The translator uses the package specification to make the procedure names visible to the static schedule. The data stream exceptions BUFFER_OVERFLOW and BUFFER_UNDERFLOW are also made visible in the package specification by renaming the same exceptions from the PSDL_SYSTEM package.

The translator's output package is called TL and is placed into a file called TL.a. The *.a* extension is required by the Ada compiler used in the development of this system. The Ada code produced by the static scheduler must utilize the Ada **with** clause to gain visibility to the TL package and all of its procedures as shown in Figure 19.

```
with TL;
package STATIC_SCHEDULE is
  ...
end STATIC_SCHEDULE;

package body STATIC_SCHEDULE is
  ...
end STATIC_SCHEDULE;
```

Figure 19. Use of Ada **With** Clause

2. Translator/Dynamic Scheduler Interface

The interface between the translator and the dynamic scheduler is also an implicit interface. The issue of concern in this case is a matter of timing. The dynamic scheduler calls the time critical atomic operator procedures in the order specified by the static schedule. When a time critical operator finishes execution before the end of its period the dynamic scheduler will invoke a non-time critical operator during the time remaining to the end of that period. Just before the end of the period, the non-time critical operator is interrupted, if it is still executing, so the next time critical operator can be invoked. To achieve this method of control, Ada tasks are used to implement the time critical and the non-time critical operators' schedules. The Ada **priority** pragma is used to establish relative priorities among these tasks such that the time critical operators have the higher priority. [Ref. 15]

Recall that a data stream is a mechanism which allows two or more operators to communicate via a single data value. The data stream is therefore a critical section which can be accessed by several operators. Any time an operator is performing a data stream read or write operation it is inside a critical section and must not be interrupted. To ensure that a non-time critical operator is not interrupted during a data stream operation, all data stream tasks are assigned a higher priority than the time critical operators. The actual interface between the translator and the dynamic scheduler is simply an agreement of the task priority values. The non-time critical operators are assigned a task priority value of one, the time critical operators are assigned a task priority value of three, and data stream tasks have a priority value of five.

It should be noted that although the **priority** pragma is a standard feature of any validated Ada compiler, the values that may be assigned in the pragma are implementation defined and could very possibly range from zero to zero. Thus, using this pragma to assign varying priority values makes the execution support system implementation dependent.

3. Translator/User-Interface Interface

The interface in this case is a combination of solid interface specifications and assumptions. The user-interface is described in detail in [Ref. 16]. There are six issues to be dealt with in the translator's interface with the user-interface:

- Syntactically correct PSDL source code.
- No duplicated composite operator names.
- Command line invocation of the translator.
- Atomic Ada code is inserted into the PSDL source code.
- User-defined types are placed at beginning of PSDL text.
- The first PSDL operator after the user-defined types is the root operator.

a. Syntactically Correct PSDL Source Code

The translator depends on the user-interface for its input PSDL source code. The user-interface produces the PSDL source code via the syntax-directed editor and the graphical editor. The first interface issue is an assumption that the PSDL source code produced in the user-interface is syntactically correct. The translator application generated by Kodiyak includes its own PSDL lexical analyzer and parser. If the input PSDL source code is not syntactically correct, the translator will terminate abnormally and ungracefully. Since this termination is built into the code produced by Kodiyak it cannot be controlled.

b. Duplicated Composite Operators

This issue is related to the standard naming convention used in the translator/static scheduler interface. In fact, that naming convention is dependent upon this interface issue. The single-shot rule [Ref. 6:p. 25] prevents the duplication of any operator name, composite or atomic, in a single set of child operators. That is, given any PSDL operator it is guaranteed that that operator name will not duplicate any of its sibling operator names and that its children, if any, will all have unique names.

This does not guarantee that operator name duplication will not occur among different branches of the PSDL hierarchical decomposition tree. Figure 20 shows an example in which a composite operator is used twice during execution of the prototype. The single-shot rule is maintained throughout the decomposition. According to standard hierarchical scoping rules, a name can be duplicated in two different scopes and represent two different objects. If that is the case in this example, and the operator D represents one operator in B's decomposition while D represents a different operator in C's decomposition, the translator will fail. The translator will assume that both instances of operator D are instances of the same operator and it

will only produce one set of atomic operator procedures to avoid a name conflict. Thus, if D was intended to name two different operators in different scopes, only one of those operators would have been produced and the Ada code generated by the translator would be semantically incorrect. This condition cannot be allowed to occur so an assumption is made that there are no duplicated composite operator names in a PSDL source code input to the translator. This possibility for failure can be eliminated by using the same naming conventions for composite operators as for atomic operators.

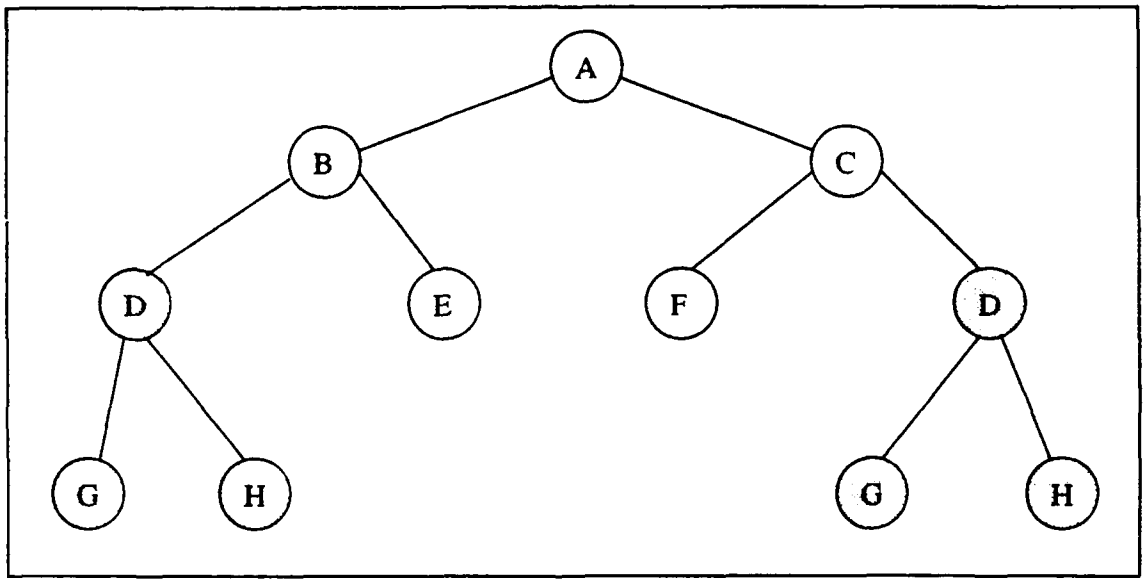


Figure 20. Duplicated Composite Operators

c. Invocation of the Translator

The translator is an executable block of code that is activated by a UNIX command line entry. There are two interface issues involved here, the name of the translator command and the set of command line options available. The translator's name is simply **translator**. To invoke the translator at the UNIX prompt type

translator [options] filename -o TL.a

where options are any of the set of legal command line options shown in Table 1 [Ref. 11:p. 206]. Filename is the name of the PSDL input source file, and -o sends the output of the translator to the file named TL.a. Normally, none of the command line

options should be selected except the -o option. See [Ref. 11] for a more detailed discussion of Kodiyak translator options.

TABLE 1. TRANSLATOR COMMAND LINE OPTIONS

-h	Print out a list of legal options.
-file	Read input from **file** rather than the standard input.
-e	Continue attribute evaluation even after an error occurs.
-l	Print out tokens as they are scanned.
-y	Print out all grammar rule reductions as they occur.
-L	Turn on LEX's debugging features.
-Y	Turn on YACC's debugging features.
-c	Generate a core image when a run-time error occurs.
-s	Print out storage statistics after all attribute evaluation is complete.
-o file	Divert the standard output to **file**.

d. Insertion of Atomic Ada Code Into PSDL Code

Originally, no provision was made for importing the actual reusable Ada code required to implement an atomic operator. The PSDL source code specified only the Ada procedure name in the software database. For example,

IMPLEMENTATION Ada filename

specifies that the atomic operator, filename, is stored in the software database and is to be used to implement this operator. The method of getting the reusable module from the software base and into the translator was not clearly defined.

It was agreed that the IMPLEMENTATION statement in an atomic operator's implementation would be exploited to make the actual Ada code visible to the translator. When the reusable module is retrieved from the software base it is inserted into the PSDL source code at the IMPLEMENTATION statement. It can easily be inserted by the retrieval system and then it can easily be extracted by the translator and placed in the TL.a package produced by the translator.

The PSDL grammar required a slight modification to the IMPLEMENTATION statement to implement this feature. It is now

IMPLEMENTATION Ada filename { source }

where source refers to the actual Ada code retrieved from the software database. The source code must be bracketed by curly braces so the translator recognizes it as a comment and not as additional PSDL code.

e. User-Defined Types

Currently, user-defined types in PSDL have not been implemented. In anticipation of their implementation, it was decided that the layout of an input PSDL program would list all of the user-defined types at the top of the input file. This is so that during translation, a symbol table can be built for the user-defined types prior to their use in the PSDL program. In addition, a user-defined type must be located prior to any instance of its use in any other user-defined type. This follows very closely to the style of other conventional block structured languages like Pascal or Ada.

f. Root Operator

Recall that a PSDL program is a decomposition of a single operator into its components. The highest level composite operator in a PSDL program is called the root operator and it represents the system being prototyped. The operators in a PSDL program are listed after all of the user-defined types and the translator is dependent upon the order of appearances of the operators. The root operator must appear before all other operators in the input file. After the root operator appears the only constraint on operator order is that any composite operator must appear before any of its subcomponent operators.

4. Translator/Software Base Interface

The translator implicitly interfaces with software base in that it expects the reusable software components to be Ada procedures. Any other type of Ada construct will cause the translator to fail. This is an implementation restriction and it is anticipated that future versions of the translator will allow various Ada constructs to be used as reusable components.

B. ADA IMPLEMENTATIONS OF PSDL CONSTRUCTS

The following sections describe the actual Ada packages and procedures used to implement each of the major PSDL constructs described in Chapter III. The data stream, exception, and timer constructs are pre-defined Ada packages and are resident in an Ada package called PSDL_SYSTEM. This package contains all of the built-in type declarations in PSDL as well as the buffer errors that can be generated. Operator constructs are not resident anywhere because they are the results of translation. The aim of the translator is to produce the Ada code which realizes PSDL operators. Appendix A contains the Ada source code for each of the implementations performed in this thesis.

1. Data Stream Implementation

Data streams are implemented as an Ada generic package containing an embedded task. Data streams are critical sections because they are referenced by several operators simultaneously, thus an Ada task is required for their implementation. Ada makes no provision for generic tasks so to get around that constraint, tasks are embedded inside of generic packages. Inside the task are the various operations that must be allowed on data streams to implement things like data triggers and data stream reads and writes.

Four separate packages were constructed to implement the four variations of data streams. They include:

- FIFO.a which implements a standard data flow stream.
- SAMPLED.a which implements a standard sampled data stream.
- FIFO_STATE_VAR.a which implements a data flow state variable data stream.
- SAMPLED_STATE_VAR.a which implements a sampled state variable data stream.

All of these data stream implementations use the same data structure shown in Figure 21 to represent the data stream. It is a simple Ada record structure consisting of two boolean flags to indicate , first, if the data stream is initialized, and second, if the data on the stream is a new value. The last field in the data stream record

is used to hold the value which resides on the data stream. This field takes the type established in the generic instantiated of the data stream package.

a. Sampled Data Streams

The GET operation first checks the data stream's **initialized** flag and if it is set it will copy the value of the data stream into the formal parameter. If the **initialized** flag is not set, then the data stream has not yet executed a write (or PUT) operation and a BUFFER_UNDERFLOW exception is raised. Anytime the GET operation successfully reads a data value, the **new_data** flag will be cleared.

The PUT operation is always successful in a sampled data stream. It will copy the value of the formal parameter into the data stream. The **initialized** flag and the **new_data** flag are always set after a PUT operation.

```
type DATA_STREAM is
   record
      INITIALIZED,
      NEW_DATA    : BOOLEAN;
      VALUE       : ELEMENT_TYPE;   -- Generic input type
   end record;
```

Figure 21. Data Stream Record Structure

The CHECK operation simply looks at the **new_data** flag and returns its value. It indicates whether or not the data stream has a new data value on it. CHECK is accessed by a function named FRESH. Ada will not allow functions inside of tasks, thus the FRESH function exists outside of the embedded task and invokes the CHECK procedure inside the task. FRESH returns true only if there is fresh data on the data stream.

b. Data Flow Data Streams

The GET operation is identical to the sampled data stream GET operation described above except that it will raise BUFFER_UNDERFLOW if the **new_data** flag is clear, that is, the buffer does not contain new data.

The PUT operation first checks the **new_data** flag and, if it is set, will raise the BUFFER_OVERFLOW exception. If the **new_data** flag is clear then the input

formal parameter value will be copied into the data stream and the **new_data** flag set.

The CHECK operation simply returns the value of the **new_data** flag. The presence of data on the data flow stream is determined by the value of the **new_data** flag. If it is true, then there exists a data value on the data stream; if it is false, then no data value effectively exists on the data stream. The FRESH function is implemented here in the same manner as it is in the sampled data stream implementation. It is the FRESH function which actually accesses the CHECK operation.

c. State Variable Data Streams

State variable data streams are almost identical to their standard data stream counterparts. A state variable data stream must be instantiated with an initial value and because of this, its **initialized** and **new_data** flags will be initialized to true. Other than this exception, a sampled state variable and a data flow state variable data stream are identical to a standard sampled and data flow data stream, respectively.

An alternative method of implementing the state variable data stream is to use the normal data streams with no changes. The initial value of the state variable could be placed in the data stream by a PUT operation prior to the execution of the prototype. This method was not used in this thesis because it would require slightly more evaluation to be performed by the Kodiyak when building the translator.

2. PSDL Exception Implementation

The Ada enumerated data type was selected to represent the PSDL_EXCEPTION data type. The PSDL_EXCEPTION type actually does not exist until compile (or translation) time. The translator is designed to scan the entire PSDL input program and collect all of the declared PSDL exceptions. Once the exception names are collected, the translator constructs a type declaration for the PSDL_EXCEPTION type. The type declaration looks like

type PSDL_EXCEPTION **is** (name1, name2, ... ,nameN);

where name1 ... nameN are the unique exception names used in the input PSDL program. The PSDL_EXCEPTION type is declared so that it is visible to all of the operator procedures constructed by the translator.

The implementation of a data stream to be used as an exception data stream is not constructed in this thesis. A firm definition of PSDL exceptions and their semantics has not been reached. The ideas presented in this thesis concerning PSDL exceptions are still only concepts which have not been realized.

To implement exception data streams, a package similar to the FIFO.a implementation is required. The **new_data** flag can be used to indicate if an exception is raised or not. A set flag implies a raised exception; a cleared flag implies a cleared exception. The PUT and GET operations should also be modified so they do not generate any buffer errors.

An exception cannot overwrite another exception value that is currently active, or raised. When the PUT operation is invoked on an exception data stream that is carrying an active exception, it should discard the new exception value and return normally. The concept being enforced here is that the first exception generated has highest priority on a data stream until it is cleared. A data stream exception is cleared when the GET operation is performed.

The FRESH function should be replaced with a function called ACTIVE. The *ACTIVE function takes a single argument, a PSDL_EXCEPTION type, and returns* true if the argument is currently in the data stream and has not been read.

3. PSDL Timer Implementation

Timers are implemented as a generic package containing the timer data structure and the operations necessary for the timer state machine described in Chapter III. The timer data structure is shown in Figure 22 and contains four fields:

- START_TIME. Time when the last start operation was initiated.
- STOP_TIME. Time when the timer was last stopped.
- ELAPSED_TIME. Elapsed time accumulated up to the last stop operation.
- PRESENT_STATE. The present state (running, stopped, or initial) of the timer state machine.

The timer package uses the Ada CALENDAR package to gain access to the system clock and built-in time manipulation arithmetic such as subtracting the start time from the system clock's current time to get an elapsed time. The timer package

expects all time values to be in milliseconds since that is the default in PSDL. The translator automatically converts all time expressions in the PSDL source code to millisecond expressions.

```
type  STATE is (INITIAL, RUNNING, STOPPED);

type TIMER is
   record
     START_TIME,
     STOP_TIME          : CALENDAR.TIME;
     ELAPSED_TIME       : DURATION;
     PRESENT_STATE      : STATE := INITIAL;
   end record;
```

Figure 22. PSDL Timer Data Structure

4. PSDL Operator Implementation

Unlike the other major PSDL constructs, PSDL operators do not have a standard generic package ready to be instantiated. The implementation of operators is not a straight-forward process that can be performed by a single piece of Ada source code. An operator is a set of PSDL constructs and a set of PSDL control constraints that dictate how to manipulate the PSDL constructs in the current operator, as well as those constructs in any higher-level operators that are visible, like timers. The implementation of the PSDL operator is the translator's single most important function.

The design of a PSDL operator implementation was driven by the semantics of PSDL and by the needs of the static scheduler. The PSDL semantics determined the use of Ada control structures to implement PSDL control constraints. The static scheduler drove the choice of structure for an Ada implementation of a PSDL operator. The static scheduler requires a set of simple procedures which can be called in a specific order to effectively execute the prototype. An added requirement of the static scheduler is that the procedures must also be executed according to a set of timing constraints. The static scheduler must have total control over the execution of an

atomic operator. Allowing the static scheduler to call a composite operator procedure which then calls its own subcomponent procedures implies that necessary timing constraints and operator precedence constraints would be determined and implemented by the translator itself. It would have to ensure that such constraints were built-in to the translation.

Doing this obviously defeats the purpose of the static scheduler and places an unnecessary requirement on the translator. Therefore, the static scheduler must determine all timing and precedence constraints and apply them to only the atomic operators in a PSDL program. Only by scheduling the atomic operators can the static scheduler be guaranteed full control over timing constraints. The requirement placed on the translator, then, is to produce a set of stand-alone procedures that the static scheduler can schedule individually and have confidence that only a single atomic operator will be executed at a time.

Meeting that single requirement is a key issue in this thesis. Doing so means that a PSDL specification which is hierarchically structured must be broken down into a set of stand-alone procedures having a flat structure. The problem with a flat structure is that normal visibility and scoping rules associated with a hierarchical structure no longer exist, yet there are rigid visibility rules that must be maintained in a PSDL operator. To meet this requirement, it was decided that a composite operator would be translated into a specification package containing all local PSDL construct instantiations and lines of visibility to all parent operator constructs. The lines of visibility are accomplished through the use of the Ada **with** and **renames** constructs.

This method is best explained by example. Appendix B is a sample PSDL system which performs no particular function. The example consists of an enhanced data flow diagram, the corresponding PSDL source code, and Ada translation. For brevity, only the shaded portions of the enhanced data flow diagram will be discussed. Operators C1, C2, and C3 are composite operators while A1, A2, A3, and A4 are atomic operators. Operators C2 and C3 are subcomponents of C1; operators A1 and A2 are subcomponents of C2; and operators A3 and A4 are subcomponents of C3. Beginning at the root, the data streams visible in operator C1 are its local data streams a, b, c, and d. PSDL rules of visibility say that all of the data streams visible

in C1 are also visible in its subcomponents, C2 and C3. Examination of the PSDL specification of operator C2 reveals that C2 requires visibility of data streams a, b, c, and d in its parent, and a local data stream, e. Likewise, operator C3 requires visibility to data streams a, b, c, d, and its own local data stream f. The atomic operator A1 requires visibility to data streams a and e; atomic operator A3 requires visibility to data streams c and f.

During translation, a specification package would be generated for each of the composite operators and in this example they would all contain nothing but data stream package instantiations because data streams are the only type of PSDL constructs used in this example. The specification packages for operators C1, C2, and C3 are shown in section C of Appendix B. All the static scheduler is concerned with at this time is the set of procedure drivers used to implement only the atomic procedures. In this example procedures C2_A1 and C3_A3 are two of those procedures. (The names used in this example are the actual names produced by the translator. Naming conventions are described in Chapter IV, section A.)

Procedure C2_A1 would be activated when the static scheduler wants to invoke atomic operator A1. Recall that operator A1 must have access to data stream e in its parent operator C2, and it must access data stream a in operator C1. To do this procedure C2_A1 must use the Ada **with** clause,

with C2_SPEC;

to gain visibility to everything that operator C2 can access. Notice that package C2_SPEC also uses a **with** clause,

with C1_SPEC;

so that it may gain access to all of its parent operator's data streams. Using the Ada **with** clause in this manner is the first step taken to establish a line of visibility from an atomic operator to its desired PSDL constructs.

The second step is in the use of the Ada **renames** clause. Looking at procedure C2_A1, the statement

DSa.GET(TL_a);

is a data stream procedure call. It specifies that the value of variable TL_a is to be written onto data stream DSa. DSa is a package and GET(TL_a) refers to a

procedure call within that package. But, which package does DSa really refer to? Because there is no DSa package declared inside procedure C2_A1, Ada will attempt to look elsewhere for the package. The only place to look is in the C2_SPEC package because of the **with** clause associated with C2_A1. Inside package C2_SPEC, a package named DSa is indeed found, however, that DSa package is part of the **renames** clause,

package DSa **renames** C1_SPEC.DSa;

That means any reference to the DSa package inside of package C2_SPEC is actually a reference to the DSa package inside C1_SPEC. Following that link up to C1_SPEC, we see that the DSa package inside C1_SPEC is the original data stream that was declared in composite operator C1. Thus, the combination of Ada **with** and **renames** clauses has created a link, or line of visibility, from the actual data stream manipulation inside procedure C2_A1 up to the ancestor operator where the data stream actually resides. The overall result is that procedures C2_A1 and C3_A3 exist as stand-alone procedures in a flat structure that can be called in any order desired by the static scheduler. It can be guaranteed that each of those procedures has access to all of its required PSDL constructs no matter where those constructs exist in the hierarchical PSDL specification.

V. TRANSLATOR CONSTRUCTION

A. MODIFYING PREVIOUS WORK

After the implementations of data streams, timers, and exceptions were completed, attention was turned toward producing an implementation of PSDL operators using Kodiyak. This was the most significant task performed in this thesis. Previous work by Moffitt [Ref. 17] and Janson [Ref. 10] produced an incomplete Kodiyak program upon which this translator was based. Moffitt's Kodiyak program was capable of parsing only PSDL programs with no boolean or arithmetic expressions.

The PSDL grammar was designed without any syntactic rules for expressions. It was assumed that expression implementations would be dependent upon the underlying language used for the PSDL translation and therefore the syntactic rules for expressions in the underlying language could be inserted in the PSDL grammar. Moffitt's Kodiyak program implemented only the general PSDL grammar in Luqi's dissertation with no capability for parsing expressions.

Implementing the Kodiyak translator in this thesis began with modifying Luqi's PSDL grammar so that it would include limited Ada expressions. Appendix C contains the updated PSDL grammar used in this thesis. Only boolean expressions were included in the updated version of PSDL because implementing the entire range of Ada expressions in the Kodiyak proved to be too time consuming.

Once the grammar was completed, Moffitt's Kodiyak was modified to accommodate the new grammar. The process of modifying and installing the PSDL grammar was performed in conjunction with Marlowe and the static scheduler because the static scheduler uses Kodiyak as well to translate PSDL into a form consistent with its needs. At this point a version of Kodiyak existed which could successfully read and parse a PSDL program, however, it performed a null translation. By performing a null translation, the Kodiyak program was effectively a template whose attributes and attribute equations could be modified as needed to produce any desired translation of PSDL.

B. THE KODIYAK PROCESS

Chapter II provided a brief overview of the Kodiyak translator generator and its purpose. Kodiyak is used in this thesis to convert a PSDL program into an abstract syntax tree (AST). Once in a tree structure the PSDL program can be scanned for information about its hierarchical structure and its functionality. Two complete passes down and up the AST are required to produce an Ada translation of a PSDL program.

Information about a PSDL program is stored in a Kodiyak map data type. The map type is the only high level data type offered in Kodiyak. The map type consists of a finite set of pairs which describe a function. A map pair consists of a key and an image such that the key is mapped into the image [Ref. 11:p. 12]. Maps are used in the translator to map a PSDL operator name into information about its structure, content, and context much like a symbol table is used in a compiler. Some of the items stored in maps include:

- Whether an operator is composite or atomic.
- The parent of an operator.
- An operator's input and output streams.
- Data stream types.

Figure 23 shows an example of an AST based on the PSDL grammar productions:

start -> psdl

psdl -> component psdl | null

component -> operator | data_type.

This small subset of the PSDL grammar is suitable for providing a high level view of a psdl program structure in an abstract syntax tree.

Movement through the AST is described in terms of passes. A pass through the AST consists of either beginning at the root of the tree and proceeding down until all of the leaves have been visited, or beginning at the leaves of the tree and proceeding up to the root. In either case a pass is a one way trip through the tree.

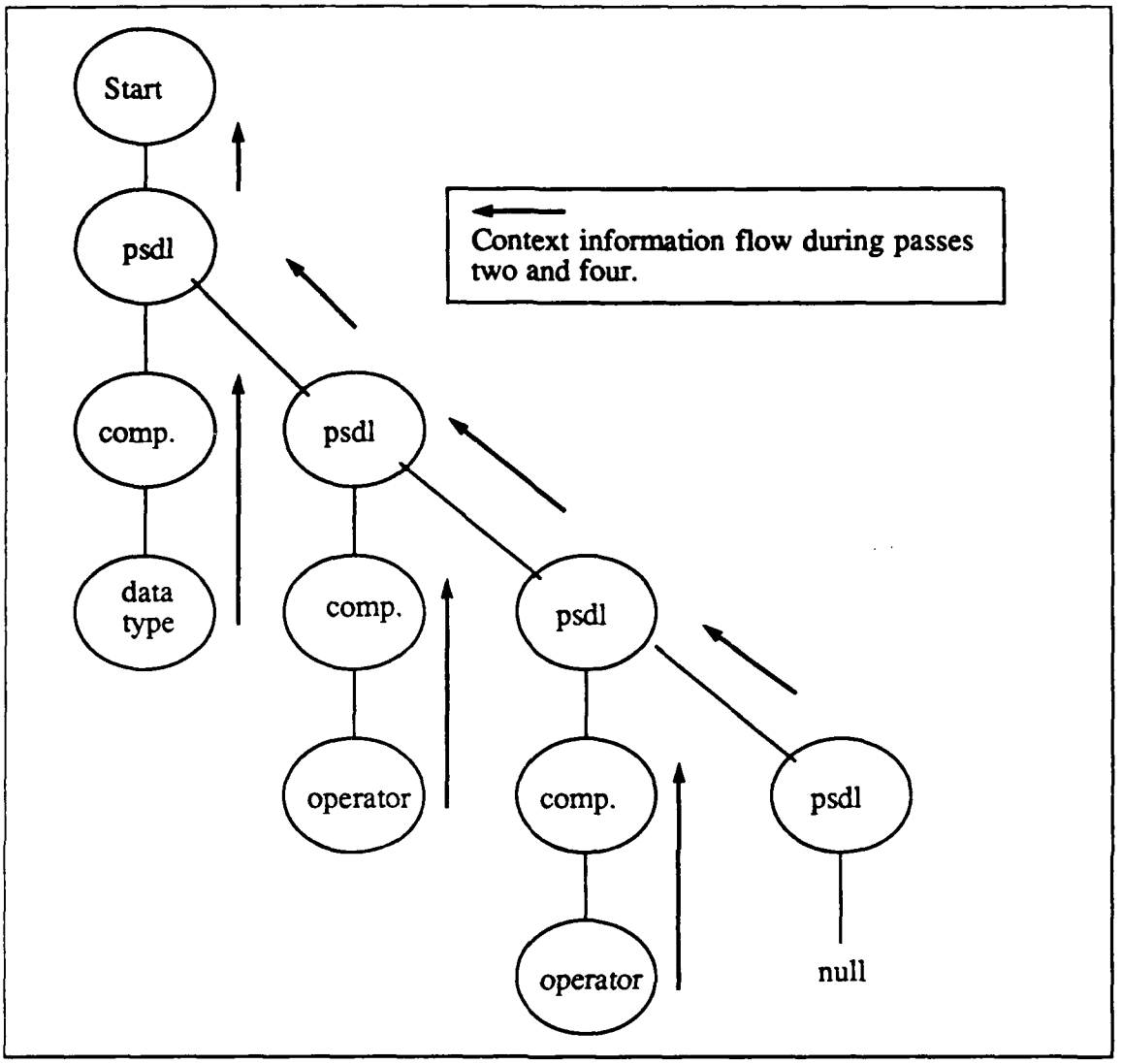


Figure 23. PSDL Abstract Syntax Tree

Pass one of the translator begins at the root of the AST, labelled START in Figure 23, and proceeds down the tree in a depth-first order. The first PSDL module in the tree is a user-defined data type. Kodiyak will scan that module first, gathering information about the data type and storing that information in a map somewhere in the DATA_TYPE node. The second PSDL module in the tree is a PSDL operator. Kodiyak will scan this module just as it did for the user-defined type, storing all of the pertinent data about the operator in a map inside the OPERATOR node. The first

pass is complete when the last PSDL node has been scanned. Some of the useful information collected at this point includes:

- Every operator name and its parent.
- Every data stream name, type, and classification as a sampled or data flow stream.
- All exceptions declared in the program.

However, all of this information is still segmented. The information in any single OPERATOR node is known only inside that node. To be useful for translation, all of the individual information maps distributed throughout the tree must be collected and their contents must be merged into a single map.

Pass two begins at the bottom of the tree in the leaves and proceeds up. The very last PSDL node in Figure 23 is an empty node. It is a result of the empty production

psdl -> null

and therefore contains no information. The information collection process actually begins with the next-to-last PSDL node since the last node is essentially empty. The information maps in each of the PSDL node's children are merged into a single map in the PSDL node. The PSDL node then routes its combined information map up to its own parent PSDL node. In this manner all of the maps in each leaf node of the AST are bubbled up to the root of the AST. After the START node has collected all of the information from its children, it is left with a collection of data which represents the entire PSDL program in a single map. This map contains all of the contextual data about the PSDL program in the AST.

Pass three begins at the root of the tree in the START node. Translation can begin now that contextual information is available for the entire tree. The presence of global context data is equivalent to possessing foreknowledge about the future of the PSDL program as well as remembering its past. The global map in the START node is routed down the tree to each PSDL node. Translations are made at each node of the tree based on the data in the global map.

These translations are strings of Ada code. As translations are made, they are stored in a map in the leaves of the tree. At the end of pass three, the individual

translations of each PSDL node have been formed but they are still segmented just as the context data was at the end of pass one.

Pass four begins at the leaves of the tree and proceeds up to the START node. Now the individual translations are gathered from each PSDL node and combined into composite groups of translated Ada code. Eventually, all of the translation information is combined in a single node at the root of the tree.

The translation information can then be placed into slots of the TL package template, which is described later. The filling of the TL package template effectively produces the TL package which is the final output from the translator. The TL package represents the Ada translation of the PSDL program in the AST.

C. TRANSLATION OF PSDL EXPRESSIONS

Before describing the overall translation process, attention should be devoted to the translation of PSDL expressions. PSDL expressions appear in a variety of PSDL statements and the translation of those statements involves the translation of expressions.

A PSDL expression is a simple boolean expression with two operands and a boolean operator. A PSDL condition is a complex PSDL expression, that is, any PSDL condition can be parsed into a group of one or more PSDL expressions joined by boolean operators. The translation of a PSDL condition involves the translation of each of its component expressions.

The PSDL grammar (shown in Appendix C) has been modified from the original grammar in [Ref. 6] to handle boolean expressions. A PSDL condition is a boolean expression dependent upon values of data streams, timers and exceptions. PSDL expressions are divided into three classes of expressions:

- Timer expressions.
- Exception expressions.
- Normal expressions.

1. Timer Expressions

Timer expressions are designed to allow two timers to be compared with each other or to allow a single timer to be compared with a constant time value. Figure 24 shows the forms of timer expressions. PSDL timer visibility rules require that a timer name used in any timer expression must be visible in the current operator or a higher level ancestor operator.

Timer constants represent a simple time value. They may be expressed as a positive integer followed by an optional time unit. Time units may be specified as milliseconds, seconds, minutes, or hours with milliseconds being the default units.

```
Timer_Expression =
      timer_name rel_op timer_name
  |   timer_name rel_op time_constant

rel_op = < | <= | > | >= | = | /=
time_constant = integer unit
unit = ms | sec | min | hours
```

Figure 24. PSDL Timer Expression

Timer expressions are translated to boolean expressions in Ada utilizing function calls into the timer packages. The generic timer package is shown in Appendix A. For example, the expression

Timer1 < Timer2

translates to the Ada expression

Timer1.READ < Timer2.READ

where the READ function returns the present value of a timer in milliseconds.

Timer expressions with constants are straight forward translations also. For example, the expression

Timer1 > 45 min

translates to the Ada expression

Timer1.READ > 2700000.

The Kodiyak program itself recognizes time units and converts all time constants in a PSDL program to milliseconds. In the example above, 45 min is translated to an equivalent value of 2,700,000 milliseconds.

When the PSDL grammar was modified to parse boolean expressions, it was patterned after the expression grammar of Ada. Therefore, PSDL follows the same precedence rules as Ada; the NOT operator has highest priority, followed by the AND and OR operations, followed by the relational operators. Parenthesized expressions will be given highest priority in any expression.

When using timer expressions, there is a semantic subtlety which the designer must be aware of. Timer expressions attempt to perform real-time operations, however, the computer system being used for development of this translator is a single-CPU machine. Expressions are evaluated left to right, sequentially. Consider the case when evaluating a timer expression like

Timer1 = Timer1.

In real-time, the expression is always true because a timer is always equal to itself. The Ada translation of this expression is

Timer1.READ = Timer1.READ

which is an expression of two calls to the READ function. The evaluation of this expression is performed sequentially, left to right. If the first function call occurs at time t, the second function call must occur at time t + d where d is at least the time required to perform the first function call. If the value of d is significantly small, the discreetization error will compensate for the time difference and the two function calls will return the same value. On the other hand, if d is sufficiently large (greater than one half of one millisecond) the values returned by the function calls will be different and the expression will always be false!

Ideally, either the Ada compiler or the translator should recognize this kind of situation and optimize it. Presently, the translator makes no consideration at all for optimization.

2. Exception Expressions

An exception expression is of the form shown in Figure 25. The purpose of PSDL exception expressions is to check for the presence of a particular PSDL

exception on a data stream. Figure 26 shows an example of an exception expression and its Ada translation. The expression evaluates to true if a PSDL exception named exceptionX is currently active on the data stream named StreamX. StreamX must be an input data stream whose type is PSDL_EXCEPTION.

```
Exception_Expression =
    Data_Stream_Name : Exception_Name
```

Figure 25. PSDL Exception Expressions

The expression is translated to a function call, ACTIVE, which returns a boolean value indicating if its PSDL exception argument is currently active. The ACTIVE function will clear the exception if it returns a true value, otherwise the exception value currently on the data stream is left alone. The use of exception expressions anywhere within an operator implicitly makes that operator an exception handler.

```
PSDL:
    StreamX : exceptionX

Ada Translation:
    EXStreamX.ACTIVE (psdl_exceptionX);
```

Figure 26. PSDL Exception Expression Example

3. Normal Expressions

A normal expression is a PSDL expression involving only normal data streams or constants. The form of a normal expression is shown in Figure 27 along with some examples of normal expressions.

The stream names used in any PSDL expression must be visible to the atomic operator containing the expression. The translator presently has no means of performing its own visibility checking or type checking in a PSDL program. It is anticipated that a static semantic analyzer for PSDL could be constructed to perform such

checks prior to sending a PSDL program to the translator. Such a tool could guarantee static correctness of the PSDL input to the translator.

```
Normal_Expression =
      data_stream_name rel_op data_stream_name
  |   data_stream_name rel_op constant
  |   constant rel_op data_stream_name

rel_op = < | <= | > | >= | = | /=
```

Figure 27. PSDL Normal Expression

The translator can only handle scalar data types in normal PSDL expressions. This is a constraint imposed by the limited expression parsing capability given to the PSDL grammar in this thesis. The user must ensure that data streams and constants used in PSDL expressions are type compatible because their translations are subject to the strong typing of Ada. The addition of user-defined types in future versions of the translator will require modifications to PSDL's expression handling capabilities.

Currently, the Ada translations of normal expressions consist of Ada variables and constants. The translator will declare a variable for each data stream used in a normal expression. Variables are required because atomic PSDL operators consume their input data stream values once during execution, however, data stream values can be used any number of times in the same operator. The data stream value is stored in a variable so that it may be referenced as often as needed without having to read the data stream more than once.

Figure 28 shows three examples of normal expressions that may be found in any PSDL condition. Translation A in Figure 28 shows the data stream variable declarations that would be generated to accommodate the expression translations shown in translation B. The variables are declared using the same type of their corresponding data streams. If Stream1 were a data stream of type FLOAT, its associated variable, TL_Stream1, would be declared as a FLOAT variable.

Note the naming convention used to name data stream variables. It concatenates the string "TL_" to the front of the data stream name. This is done to avoid naming conflicts.

The three basic types of PSDL expressions described in this section are the building blocks of PSDL conditions. Since all three expression types produce boolean results, they can be used together in a single PSDL condition as in the statement

OUTPUT a IF (a > 0) AND (t < 5 min) OR (excp : no_data)

which uses an expression of all three types.

PSDL: Stream1 < Stream2 Stream1 > 10 10 = Stream2
Ada Translation: TL_Stream1 < TL_Stream2 TL_Stream1 > 10 10 = TL_Stream2

Figure 28. Normal Expressions and Their Translations

D. THE TL PACKAGE

The final output of the translator is an Ada package named TL. A template description of the TL package is shown in Figure 29. The TL package embodies all the Ada code necessary to simulate the atomic operators in a PSDL program. It is designed to provide visibility to all of the atomic operator drivers and to the Ada exceptions which are to be handled by the dynamic scheduler/debugger. All other implementation specifics are hidden from the static and dynamic schedules.

The Kodiyak translator generator [Ref. 11] was used to produce an executable program which translates an input PSDL file into the TL package. The source code used for the Kodiyak program is contained in Appendix D and is quite lengthy. It is not the intention of this thesis to describe the Kodiyak program line-by-line. Rather,

```
package TL is

    type PSDL_EXCEPTION is (psdl_excp1, psdl_excp2, ... , psdl_excpN);
    excp1, excp2, ... , excpN  : exception;

    procedure atomic_driver1;
    procedure atomic_driver2;
    ...
    procedure atomic_driverJ;

end TL;


with PSDL_SYSTEM;
use PSDL_SYSTEM;

package body TL is

    atomic procedures drawn from software base

    PSDL operator specification packages

    PSDL atomic operator driver procedures


end TL;
```

Figure 29. TL Package Template

the concepts of the translation process will be discussed by describing the formation of the TL package.

The Kodiyak program itself was constructed incrementally by selecting a small section of the TL package and programming a Kodiyak program to produce the translation for that section. Once the Kodiyak program could produce that section, a new section of the TL package was selected and the existing Kodiyak program was modified to produce the new section as well as the previous one. This incremental process continued until a final Kodiyak program existed which could produce the whole TL package. The TL package is divided into five major sections:

- Exception declarations.
- Atomic operator driver headers.
- Atomic procedures.
- PSDL operator specification packages.
- PSDL atomic operator driver procedures.

These five sections represent the portions of the TL package which are derived from the PSDL input program. The exception declarations section defines the PSDL_EXCEPTION data type and all of the PSDL exceptions that may be raised in the PSDL program. The atomic operator driver headers section lists the procedure names which are to be called by the static and dynamic schedules. This section is a requirement of Ada packages to ensure visibility into the actual atomic operator drivers in the package body. The atomic procedures section contains all of the Ada procedures drawn from the software base. The PSDL operator specification packages section contains all of the composite operator specifications. These specifications are represented as Ada packages containing only data stream and timer instantiations. The last section contains all of the procedures used to implement the atomic operator drivers. It is these procedures that are called by the static and dynamic schedules.

1. PSDL Exception Declarations Section

This section of the TL package declares the PSDL_EXCEPTION type and all of the PSDL exceptions that are found in the PSDL input program. The EXCEPTIONS statements in all of the PSDL specifications are used to generate the

translation to place in this section. An example of an EXCEPTIONS statement is shown in Figure 30 along with the translations that are derived from it.

The EXCEPTIONS statement declares all the Ada exceptions that can be raised by an atomic operator. These exceptions must be captured by the ESS and converted to PSDL exceptions.

Exception names in a single EXCEPTIONS statement may not be duplicated, however, an exception name may be included in any number of different EXCEPTIONS statements throughout a PSDL program. Recall that exception names are unique in PSDL, so that references to the same name in two or more EXCEPTIONS statements actually refer to the same PSDL exception.

```
PSDL:
    EXCEPTIONS excp1, excp2, ...
```

```
Ada Translation A:
    type PSDL_EXCEPTION is (psdl_excp1, psdl_excp2, ...);
Ada Translation B:
    excp1, excp2, ... : exception;
```

Figure 30. EXCEPTIONS Statement Translation

There are two parts to the translation of the EXCEPTIONS statement. The first part, translation A in Figure 30, places the exception names into a map. This map will be combined with the exception name maps of other atomic operators to generate a single list of PSDL exception names. To avoid naming conflicts, each exception name in the EXCEPTIONS statement is prepended with the string "psdl_" when placed into the map. At the end of pass four in the translator, the combined exception names are used to generate the PSDL_EXCEPTION type declaration as shown in Figure 30. This type is an Ada enumerated data type where the enumerated items are the unique exception names retrieved from the map.

Translation B in Figure 30, generates the Ada exception declarations. These are the actual exceptions raised by the atomic components from the software base. The names in the EXCEPTIONS statement are not altered when translated into Ada

exception declarations. Like their PSDL counterparts, Ada exceptions used in a PSDL program are unique. Multiple references to the same exception name refer to the same exception regardless of location in the PSDL source code.

2. Atomic Operator Driver Headers Section

This section is a requirement of Ada, not of PSDL or CAPS. Visibility to the objects contained inside an Ada package body are determined by the package specification [Ref. 18:p. 7-1]. It is desired to provide visibility only to the exceptions which could be generated by a PSDL program and to the atomic operator drivers. The statements in this section are simply the procedure headers of the atomic operator drivers with no bodies.

The atomic operator driver headers are produced during the third pass of the Kodiyak program. Figure 31 highlights the portions of the PSDL program which determine these translations. When processing the PSDL statement,

OPERATOR Sub_Operator_Name TRIGGER Triggering_Condition,

the contextual data from pass two is used to identify Sub_Operator_Name as an atomic operator or composite operator. If it is atomic, a procedure header will be generated for it by concatenating the strings "procedure", Parent_Operator_Name, "_", Sub_Operator_Name, and ";". Parent_Operator_Name is known from the OPERATOR statement in the PSDL specification section. The concatenated string is stored in a map. If there are any more atomic subcomponents in the current operator, their procedure headers will be stored in the map also. If Sub_Operator_Name is a composite operator, no procedure header will be generated because composite operators are not executable and therefore do not require a driver.

During pass four of the Kodiyak, all of the atomic operator driver headers are collected into a single string which is then placed into the atomic operator driver headers section of the TL package.

3. Atomic Operators Section

This section of the TL package contains a set of Ada procedures which are the reusable components drawn from the software base. The user interface inserts these procedures into a PSDL program before it is sent to the translator. Figure 32 highlights the PSDL statement used to derive this section.

Translation of Atomic_Source is more of a cut and paste operation than a translation process since Atomic_Source is already in Ada code. During pass one, Source_Code is collected in its entirety and placed into a map. Pass two then collects all of the various Atomic_Sources in the PSDL program and combines them into a single map item. The translation is effectively complete at the end of pass two.

The single map item representing all of the atomic procedures as a whole is placed into the atomic operators section of the TL package at completion of pass four.

```
OPERATOR Parent_Operator_Name
SPECIFICATION
    ...
END

IMPLEMENTATION GRAPH
    Link Statements
    DATA STREAM ...
    TIMER ...

CONTROL CONSTRAINTS
    OPERATOR Sub_Operator_Name
    TRIGGER Triggering_Condition
    CONSTRAINTS

END
```

Figure 31. Statements Used to Build Atomic Procedure Header

4. PSDL Operator Specification Packages Section

This section contains a set of Ada package specifications. There is an Ada package specification for every composite operator in a PSDL program. Each specification package can be represented by a template with various slots. The specification package template in Appendix E was derived to guide the development of the Kodiyak program. It is a template of an Ada package specification with slots to fill in.

```
OPERATOR Atomic_Operator
SPECIFICATION
    ...
END
    IMPLEMENTATION ADA Atomic_Operator {Source_Code}
END
```

Figure 32. Statements Used to Build Atomic Operator Header

The translator must map certain PSDL statements into the slots in the specification package template. The slots in the specification package template represent data stream and timer instantiations. Since these packages only serve to declare data structures, no accompanying package bodies are required. The slots of the specification package template are:

- Package header slot.
- Input / output stream instantiations slot.
- State variable data stream instantiations slot.
- Local data stream instantiations slot.
- Local timer instantiations slot.

a. Package Header Slot

This slot names the current operator's package and its parent operator's specification package. The Ada code which fills the slot is formed by translating the PSDL statements shown in Figure 33.

There are two ways to translate a PSDL operator specification header into the specification package header. Translation A in Figure 33 shows the most common translation, it contains an Ada **with** clause. Parent_Operator_SPEC is the name of the specification package belonging to the parent of Current_Operator. By "withing" the parent specification package, the current operator is given a line of visibility into it so that the current operator may access any external PSDL constructs in its parent operator.

The name given to the specification package being formed is simply the name of the current composite operator, Current_Operator in this example, concatenated with the string "_SPEC".

Translation B is a special case and only applies to the root operator. The root operator has no parent, therefore there is no need to use a **with** clause. The only translation to perform is the package name which is identical to the procedure just described in Translation A.

```
PSDL:
        OPERATOR Current_Operator
        SPECIFICATION
-------------------------------------------------
Ada Translation A:
        with Parent_Operator_SPEC;
        package Current_Operator_SPEC is
Ada Translation B:
        package Current_Operator_SPEC is
```

Figure 33. Operator Specification Header Translation

b. Input /Output Stream Instantiations Slot

This slot contains data stream package instantiations corresponding to the data streams listed in the INPUT and OUTPUT statements of a PSDL operator specification. Figure 34 shows an example of each and their translations.

The INPUT and OUTPUT statements describe the external inputs and outputs to and from an operator. To the translator, they indicate the particular data streams visible in the parent operator which should also be visible inside Current_Operator.

The names used in the INPUT and OUTPUT statements must replicate names used in the parent operator. For example, if a parent composite operator owns two data streams named X and Y, and an atomic child operator wants to use X as an input stream and Y as an output stream, then the atomic child must use the names X and Y in its INPUT and OUTPUT statements. If a data stream name is used in an

INPUT or OUTPUT statement which is not used in the parent operator, then a semantic error results which will cause the translator to rename a package that does not exist.

The translation of an INPUT or OUTPUT statement is simply a renaming of a data stream package which can be found in the parent operator's specification package. All data stream packages are named by prepending the letters "DS" to the stream names listed in the INPUT and OUTPUT statements.

It was decided during the design of the translation, that an INPUT or OUTPUT statement used in the root operator would be semantically incorrect. The root operator represents the highest level operator in the system, in fact it is the system. As such, there are no external operators to communicate with the root, therefore there should be no external inputs or outputs to or from the root.

```
PSDL:
    INPUT Stream1, Stream2        : type_name
    OUTPUT Stream3, Stream4       : type_name

Ada translation:
    package DSStream1 renames Parent_Operator_SPEC.DSStream1;
    package DSStream2 renames Parent_Operator_SPEC.DSStream2;
    package DSStream3 renames Parent_Operator_SPEC.DSStream3;
    package DSStream4 renames Parent_Operator_SPEC.DSStream4;
```

Figure 34. INPUT, OUTPUT Statement Translations

c. State Variable Data Stream Instantiations Slot

This slot contains the Ada code required to instantiate state variable data streams. The STATES statement in a PSDL operator specification is the source of translation. Figure 35 shows an example STATES statement and its translation.

The STATES statement declares local internal state variables inside Current_Operator. To the translator, these are new state variable packages which must originate inside Current_Operator's specification package.

The names used in the STATES statement must not duplicate any data stream names used in an INPUT, OUTPUT, or DATA STREAM statement. Such an occurrence would cause the translator to generate two data stream packages with the same name in the same specification package. The Ada compiler will not allow this.

The items in initial_value_list must be scalar constants compatible with type_name. Arithmetic constant expressions are not allowed in this version of the translator because arithmetic expressions are not a part of the PSDL grammar. Future versions may implement this feature. The number of initial values in the initial value list must be at least one. The first initial value will be associated with the first state variable; the second initial value is associated with the second state variable, and so on. If there are more initial values than state variables, the extra initial values are ignored. If there are fewer initial values than state variables, the last initial value in the list is assigned to all state variables without a corresponding initial value. For example, the statement

STATES A,B,C,D : INTEGER INITIALLY 0,1

would assign a value of zero to state variable A and a value of one to state variables B, C, and D.

State variables are translated into new instantiations of state variable data streams with an initial value as shown in Figure 35. Because they are data streams, the standard data stream naming convention of prepending "DS" to the stream names in the STATES statement is used. Whether the state variable is instantiated as a sampled or data flow data stream depends on the use of the data stream in lower level operators. Recall that only data streams used in a **BY ALL** trigger condition are data flow streams, all others are sampled streams. Since the classification of the data stream is dependent upon its context, this translation cannot be performed before pass three of the translator.

d. Local Data Stream Instantiations Slot

This slot contains the Ada code required to instantiate new data streams which are local to the current operator. These instantiations are translations of the DATA STREAM statement found in a PSDL operator implementation. Figure 36 shows an example of a DATA STREAM statement and its translation.

PSDL:

```
STATES Stream5, Stream6 : type_name INITIALLY initial_value_list
```

Ada translation:

```
package DSStream5 is new SAMPLED_STATE_VAR (type_name,initial);
package DSStream6 is new DATAFLOW_STATE_VAR (type_name,initial);
```

Figure 35. STATES Statement Translation

The purpose of the DATA STREAM statement is to declare local data streams in a composite operator which may be used by its subcomponent operators. The rules for input, output, and state variable data stream names apply to local data streams as well. The names declared in a DATA STREAM statement cannot duplicate any other data stream names declared in a higher level operator, i.e., there cannot be two different data streams with the same name in a PSDL program.

PSDL:

```
DATA STREAM Stream1, Stream2, .. : type_name
```

Ada Translation:

```
package DSStream1 is new SAMPLED_STREAM(type_name);
package DSStream2 is new DATAFLOW_STREAM(type_name);
```

Figure 36. DATA STREAM Statement Translations

A DATA STREAM statement is translated into a set of data stream package instantiations. Because the data streams are local, they must be instantiated using the Ada **new** statement instead of the **renames** statement used for data streams in INPUT and OUTPUT statements. Whether to declare the data stream package as a sampled or dataflow stream is dependent upon the stream's context. The context is not known until pass two of the translator and the translation is not performed until pass three.

e. Local Timer Instantiations Slot

This slot in the specification package template contains the Ada code required to instantiate timer packages. Figure 37 shows an example of a TIMER statement and its translation.

The Ada code in the timer instantiations slot comes directly from the translation of the TIMER statement. The translation itself is performed during pass three of the translator, however, it could just as well have been performed in the first pass because there is no dependence at all upon the context of the timer.

To avoid naming conflicts, the translator prepends the string "TM_" to all timer names as shown in Figure 37.

After all of the slots in the specification package template have been filled with Ada code, they are concatenated together to form a single Kodiyak string which is the specification package. That string is placed into a map and will be collected and combined with all of the other specifications packages in the PSDL program. This final collective group is placed into the PSDL operator specification packages section of the TL package template.

PSDL:

```
TIMER t1, t2, ...
```

Ada Translation:

```
package TM_t1 is new PSDL_TIMER;
package TM_t2 is new PSDL_TIMER;
```

Figure 37. TIMER Statement Translation

5. PSDL Atomic Operator Driver Procedures Section

This section of the TL package contains a set of Ada procedures which act as drivers for the procedures drawn from the software base. Each of the procedures in this section implements an atomic operator in terms of its PSDL control constraints. These procedures are constructed during passes two, three, and four of the Kodiyak program. A driver procedure must be built when the Kodiyak is processing a

composite PSDL operator which has atomic subcomponents. It is in this situation when the Kodiyak has access to the information required to build the driver procedure. That information includes:

- The atomic operator name.
- The parent operator name.
- The atomic operator's control constraints.

The construction of a driver procedure can be patterned after the implementation template shown in Appendix E. The implementation template is a partial Ada procedure with seven slots to be filled in. The translator must map certain PSDL statements into each of these slots to perform a successful translation. The slots to be filled in are:

- A header slot.
- A data stream variable declaration slot.
- A triggering condition slot.
- An atomic procedure call slot.
- A control constraints slot.
- An unconditional output slot.
- An exception closure slot.

Following is a description of how various PSDL statements are interpreted and translated into Ada code in the implementation template.

a. Header Slot

Slot one is the header slot. To fill this slot, the translator must generate an Ada **with** clause, and Ada **use** clause, and the procedure name. The **with** and **use** clauses import the parent operator's specification package. This gives the atomic operator driver access to any data streams and timers that were visible in the parent operator. The procedure name of the atomic operator driver is constructed by concatenating the parent operator name and the atomic operator name with an underscore character between them.

The translation required to fill this slot is performed during pass three of the Kodiyak. The set of PSDL statements used for the translation are highlighted in Figure 31. The upper highlighted statement specifies the name of the parent operator;

the lower statement names the atomic subcomponent. If the subcomponent in the lower statement is a composite operator, no translation is performed for it. Composite operators are not executable, therefore they do not have an atomic procedure for which a driver must be constructed. The **with** and **use** statements in the header slot are formed by concatenating the parent operator name in the upper highlighted statement in the figure with the string, "_SPEC;". Using the generalized PSDL operator in Figure 31, the translation of the highlighted statements appears as

```
with Parent_Operator_SPEC;
use Parent_Operator_SPEC;
procedure Parent_Operator_Name_Sub_Operator_Name is
   ...
end Parent_Operator_Name_Sub_Operator_Name;
```

b. Data Stream Variables Slot

Slot two is the data stream variables slot. This slot declares Ada variables which will be used to physically interface an atomic operator with its data streams. The statements which contribute directly toward the translation of the data stream variables are the INPUT and OUTPUT statements in an operator specification. The INPUT and OUTPUT statements determine the variables needed to make a procedure call to the atomic Ada code.

Consider the INPUT and OUTPUT statements in Figure 38. If these statements belong to an atomic operator's specification, then they can be interpreted as the input and output parameters to the atomic procedure. However a, b, c, x, y, and z are all data streams in the PSDL specification but the procedure must be called with actual parameters as in

```
Atomic_Procedure_Call (TL_a, TL_b, TL_c, TL_x, TL_y, TL_z);
```

When the translator scans an atomic operator's INPUT and OUTPUT statements, the stream names in those statements and their types are stored in a map during pass two. Later, when the atomic driver procedure is being formed, those stream names can be recalled from the map along with their types and translated into variable declarations as shown in Figure 38. To avoid name conflicts, variable names are formed by prepending the string "TL_" to the data stream name.

c. *Triggering Condition Slot*

Slot three in the implementation template is to be filled with the triggering condition translation. Triggering conditions were described in Chapter III. The form of an atomic operator's triggering condition is shown again in Figure 39 along with the form of its translation. The translation of a triggering condition involves the translation of its data trigger and input guard as shown in Figure 40.

Triggering conditions are translated into Ada conditionals (If-Then statements) where the conditions to be evaluated are the Ada translations of the data trigger and input guard. The input guard is nothing more than a PSDL condition and is translated as described previously in section C. The data trigger checks for the presence of new data values on its data stream arguments. The FRESH function of the data stream implementation is used to realize a data trigger, recall that the FRESH function returns true if there is a new data value on its data stream, otherwise it returns false. Figure 40 shows that a data trigger can be a **BY ALL** or a **BY SOME** trigger. The **BY ALL** trigger requires fresh data on all of its data stream arguments. This type of trigger is translated into a series of boolean function calls joined by the Ada AND operator. The **BY SOME** trigger requires fresh data on at least one of its data stream arguments. It is translated into a series of boolean function calls joined by the Ada OR operator.

Both the data trigger and the input guard may be null conditions. In each case a null condition is translated to a boolean constant value of true so the condition will always be satisfied during execution.

Embedded within the translation of a triggering condition is a translation to read data stream values. This is a semantic requirement of PSDL; recall that if a data trigger is satisfied, a PSDL operator will consume its data stream input values. The PSDL statement required for this translation is the INPUT statement of the atomic operator's PSDL specification. Figure 41 shows an example of an INPUT statement and its corresponding translation into data stream read operations. The GET procedure of the data stream implementations are used to implement this requirement.

PSDL:
```
INPUT a, b, c : type_name
OUTPUT x, y, z : type_name
```

Ada Translation:
```
TL_a : type_name;
TL_b : type_name;
TL_c : type_name;
TL_x : type_name;
TL_y : type_name;
TL_z : type_name;
```

Figure 38. Data Stream Variable Translations

PSDL expressions are translated into Ada boolean expressions whose operands are data stream variables, timers, or exceptions. If any data stream variables are used in a PSDL operator, the data stream read translation is necessary to ensure those variables have initial values. Only the input data stream variables need to be initialized, the output data stream variables are initialized by the atomic procedure when it is called by the atomic procedure driver.

Triggering_Condition =
```
TRIGGERED Data_Trigger IF Input_Guard
```

Ada Translation:
```
if Data_Trigger_Translation then
    Read Input Streams
    if Input_Guard_Translation then
    ...
    end if;
end if;
```

Figure 39. PSDL Triggering Condition Translation

```
Data_Trigger =
      BY ALL id1, id2, ...
  |   BY SOME id1, id2, ...
  |   null

Input_Guard =
      PSDL_Condition
  |   null;
```

```
Ada Translation: (Data_Trigger)
      DSid1.FRESH and DSid2.FRESH
      DSid1.FRESH or DSid2.FRESH

Ada Translation: (Input_Guard)
      refer to section C, Translation of PSDL Expressions
```

Figure 40. PSDL Data Trigger and Input Guard Translation Forms

d. Atomic Procedure Call Slot

Slot four of the implementation template is filled with an Ada procedure call. The procedure being called belongs to the group of reusable components drawn from the software base. Each atomic procedure driver calls one of those atomic procedures. The translation of the atomic procedure call is generated from the INPUT and OUTPUT statements of the atomic operator's PSDL specification. The data streams in these statements correspond to in and out parameters in an Ada procedure. The exception to this is for data streams of type PSDL_EXCEPTION. PSDL exceptions are not permitted to be used as parameters for an atomic procedure. Exceptions raised in atomic procedures are Ada exceptions which the translator is responsible for converting into PSDL exceptions and transmitting on PSDL_EXCEPTION data streams.

When the translator scans the INPUT and OUTPUT statements of an atomic operator, the data stream names are collected into a map entry which is the parameter list to be used in the atomic procedure call. During pass three of the translator, this map entry is retrieved and converted into a procedure call when an atomic

operator is being scanned. Figure 42 shows an example of the PSDL statements required to produce an atomic procedure call. The name used for the atomic procedure call is the exact name given in the statement

OPERATOR Atomic_Name TRIGGERED ....

PSDL: INPUT x, y, z : type_name
Ada Translation: DSx.GET(TL_x); DSy.GET(TL_y); DSz.GET(TL_z);

Figure 41. Data Stream Read Translation

e. Control Constraints Slot

Slot five of the implementation template is a collection of translations of timer operations, exception operations, and conditional output operations. Timer operations can be directly translated into Ada if-then statements. The three timer operations and their translations are shown in Figure 43. The timer condition is a PSDL condition and is translated as described in section C. Timer operations are translated into procedure calls to timer packages. Examination of the PSDL grammar in Appendix C shows that the timer condition is optional. If a null timer condition is specified, the translator will interpret it as a boolean constant of value, true. The constant true value will force the Ada if-then translation of a timer operation to always execute. For example, the timer operation

START TIMER t1

will translate to the Ada if-then statement

```
if true then
   t1.START;
end if;
```

which will always execute.

```
PSDL:
    OPERATOR Atomic_Name
    SPECIFICATION
        INPUT a, b, c  : type_name
        OUTPUT x, y, z  : type_name
        ...
    END
    IMPLEMENTATION ADA Atomic_name {Source_Code}
    END
```

```
Ada Translation:
    Atomic_Name (TL_a, TL_b, TL_c, TL_x, TL_y, TL_z);
```

Figure 42. Atomic Procedure Call Translation

Exception operations are also direct translations. Figure 44 shows the forms of an Exception operation and its translation. Exception operations have only a single form, however the exception condition is optional. If an exception condition is specified, the EXCEPTION statement will be translated into an Ada if-then statement. The exception condition is a PSDL condition and is translated as described in section C. The exception operation is translated into a data stream PUT procedure call. To be semantically correct, the data stream must have been declared as a PSDL_EXCEPTION, if not, then a type mismatch will occur during compilation of the Ada translation.

The translator handles a null exception condition in the same way a null timer condition is handled. The condition_translation is a boolean constant of value, true. The Ada if-then statement representing the EXCEPTION statement will always execute if the exception condition is null.

When the concept of PSDL exceptions was being designed, it was desired to have PSDL exceptions behave as closely as possible to Ada exceptions. The Ada **return** statement shown in the translation of a PSDL EXCEPTION statement is an attempt to meet this requirement. The overall function of the EXCEPTION statement is to place a PSDL exception onto a data stream and then terminate execution of the atomic operator. The **return** statement achieves this effect.

PSDL:

```
STOP TIMER t1 IF Condition1
START TIMER t1 IF Condition2
RESET TIMER t1 IF Condition3
```

Ada Translation:

```
if Condition1_Translation then
    t1.STOP;
end if;

if Condition2_Translation then
    t1.START;
end if;

if Condition3_Translation then
    t1.RESET;
end if;
```

Figure 43. PSDL Timer Operation Translations

The final control constraint to be translated is the conditional output statement. The forms of the conditional output statement and its translation are shown in Figure 45. The translation of conditional output statement is very similar to the translations of the previous two types of control constraints. The main difference is that the output condition is not optional.

Conditional output statements are translated into Ada if-then statements, also. A list of output data streams may be specified in the statement and the translator will generate a data stream PUT procedure call for each stream in the list. The semantic restrictions on the conditional output statement are that no data stream be listed in more than one output statement; and no PSDL_EXCEPTION data streams may be listed in the output statement. Output to a PSDL_EXCEPTION data stream is accomplished in the EXCEPTION statement previously described.

Control constraints are translated in the order which they are written in the PSDL program. The PSDL grammar does not specify a standard ordering for control constraints so they may be placed in any order desired as long as they all follow the

operator TRIGGER statement. The designer should be aware of control constraint ordering when using the EXCEPTION control constraint because if the EXCEPTION statement is executed, none of the following control constraints will be executed. All of the control constraint translations are combined into a single Kodiyak string and placed into a map in the atomic operator's node in the abstract syntax tree. This map entry is then used to fill the control constraints slot in the implementation template.

```
PSDL:
    EXCEPTION excp ON Data_Stream IF Condition

Ada Translation:
    if Condition_Translation then
        DSData_Stream.PUT(psdl_excp);
        return;
    end if;
```

Figure 44. PSDL EXCEPTION Statement Translation

f. Unconditional Output Slot

Slot six in the implementation template is the unconditional output slot. The translator fills this slot with zero or more data stream PUT procedure calls. An implicit translation is made to fill the slot based on the conditional output control constraints. An atomic PSDL operator must write calculated data values to all of its output data streams sometime before it terminates. Output may be done conditionally using the conditional output statement described previously, or unconditionally. Any output data stream that is not written to in a conditional output statement must be written to in an unconditional output statement.

```
PSDL:
    OUTPUT x, y IF Condition

Ada Translation:
    if Condition_Translation then
        DSx.PUT(TL_x);
        DSy.PUT(TL_y);
    end if;
```

Figure 45. Conditional Output Statement Translation

Figure 46 shows an atomic operator named AO with output data streams named x, y, and z. Assume that in AO's control constraints there is a single conditional output statement in which data stream x is the only stream specified. The translator will generate Ada code to unconditionally output to data streams y and z at the end of the control constraints translations. The translations appear as

```
DSy.PUT(TL_y);
DSz.PUT(TL_z);
```

in the unconditional output slot in the implementation template.

g. Exception Closure Slot

Slot seven in the implementation template contains the exception closure used to capture Ada exceptions raised by the atomic operator procedure and convert them to PSDL exceptions. The exception closure is translated from the EXCEPTIONS statement in the PSDL operator specification. The purpose of the EXCEPTIONS statement is to list all of the Ada exceptions that may be raised in the atomic procedure. The translator must construct an exception closure which recognizes those Ada exceptions and converts them to PSDL exceptions. Figure 47 shows an example of the statements required to construct an exception closure.

The translator requires information gathered from the EXCEPTIONS and OUTPUT statements in an atomic operator's PSDL specification in order to construct the exception closure. When the translator scans any output data streams of type PSDL_EXCEPTION, it places their names into a map. When the EXCEPTIONS

statement is scanned, each exception name in the list is translated into an Ada exception handler statement of the form

when exception_name =>.

The translator then produces the required data stream PUT procedure calls by recalling the names of the output PSDL_EXCEPTION data streams. The example translations shown in Figure 47 would be produced for an atomic operator with output data streams named x and y, and exceptions named excp1 and excp2.

```
OPERATOR AO
    ...
    OUTPUT x, y, z : INTEGER
    ...
END

Elsewhere in AO's control constraints:
    OUTPUT x IF x > 0
```

Figure 46. Unconditional Output Example

An exception closure translation is always produced for every atomic operator regardless of the presence of an EXCEPTIONS statement. If there is no EXCEPTIONS statement in an atomic operator, an empty exception closure is produced. An empty closure simply re-raises any Ada exceptions that might be raised by the atomic procedure. The empty closure appears as

```
exception
 when others =>
raise;
```

which will trap any Ada exception and raise it again to be handled elsewhere. "Elsewhere" in this system means the dynamic scheduler.

PSDL:

```
OUTPUT x, y : PSDL_EXCEPTION
EXCEPTIONS excp1, excp2
```

Ada Translation:

```
exception
when excp1 =>
    DSx.PUT(psdl_excp1);
    DSy.PUT(psdl_excp1);
when excp2 =>
    DSx.PUT(psdl_excp2);
    DSy.PUT(psdl_excp2);
when others =>
    raise;
```

Figure 47. Exception Closure Example

The completed exception closure is placed into the exception closure slot during pass three. After the exception closure slot is filled, the translator can then combine all of the filled slots of the implementation template into a single Kodiyak string which is the atomic operator driver. The driver is placed into a map and later retrieved during pass four with all the other drivers that have been produced. The drivers are collected as a whole and placed into the atomic operator drivers slot in the TL package. The final TL package translation is then produced by combining all of the slots of the TL package template into a single Kodiyak string. The translator then writes that string into a file named TL.a and the string itself is the TL package.

VI. CONCLUSIONS

Implementing a PSDL to Ada translator is feasible. While the translator in this thesis is still only a partial implementation, it has demonstrated that there is a systematic method to translate PSDL into Ada. The template method described in this thesis provides a clear picture of the mapping from PSDL to Ada. It is a tool which can be used to analyze the syntax/semantic relations between Ada and PSDL. The Kodiyak translator generator is an excellent tool for expressing syntax/semantic relations between languages. By expressing the source language, PSDL, in terms of its abstract syntax tree it is relatively straight-forward to map branches of the abstract syntax tree into the specification and implementation templates derived in this thesis. Kodiyak was found to be very flexible concerning changes to the grammar. During design and implementation of the translator, the PSDL grammar was changed slightly. Incorporating those changes was very simple using Kodiyak. Changing the translation produced by Kodiyak was a simple process as well. In fact, the translator was constructed by incrementally programming the Kodiyak to perform a basic translation and then modifying on top of that version to perform the next basic translation.

Kodiyak does have some drawbacks, however, which were noticed during research in this thesis. Kodiyak is not a language which can be learned easily. To use it effectively requires a familiarity with the LEX and YACC tools in UNIX. This is no simple task for anything less than a seasoned programmer. Kodiyak is also an extremely resource intensive tool. Compiling the Kodiyak source code in Appendix D takes just over seven minutes on a dedicated Sun 3/60 workstation. The output of Kodiyak is an object file over 230 kilobytes in size. Perhaps the most annoying problem with Kodiyak is its lack of high level data types and operations. The map type is the only high level data type offered by Kodiyak and the operations that can be performed on it are few. Manipulating groups of objects in a map is cumbersome. There is no looping facility to allow sequencing through a group of similar objects as can be done in an array in Pascal or Ada. The lack of high level data types and operations

forces the programmer to exert more effort to manipulate data structures in a Kodiyak program.

Further work is necessary to make the implementation of the translator complete. A static semantic analyzer is necessary. Its purpose would be to accept a PSDL program from the user interface system, perform a thorough semantics check on it, and if all is well, send the syntactically and semantically correct PSDL program to the translator and dynamic scheduler. Type checking, timing constraint analysis, and name checking are the primary functions required. Currently, type checking is not particularly difficult, but when user-defined data types are eventually incorporated into the translator, type checking will become a serious task. Simple timing constraint analysis of PSDL operators should be performed before sending a PSDL program to the static scheduler. This process ensures that a composite operator's period or maximum execution time is sufficient to allow its subcomponents to fire comfortably. Name checking would ensure that composite operator names are not duplicated in a PSDL program. This is vital to the translator and static scheduler to guarantee that all of the procedure names generated by the translator are unique.

The translator, itself, still requires some enhancements to be fully operational. PSDL allows the software designer to construct his own user-defined data types in PSDL. The translator in this thesis does not translate user-defined types, all efforts were devoted to PSDL operators only. PSDL exceptions are not complete. This thesis has developed a conceptual design for PSDL exceptions and their semantics, however they are not implemented in the translator. Enhancements can also be made to improve the visibility of PSDL timers, data streams, and operators. The present implementation depends on all names of these constructs being unique throughout a PSDL program. Ideally, name visibility should conform to hierarchical visibility rules.

CAPS is dependent upon an extended compiler technique for executing PSDL programs. This technique is a two-part process. A PSDL program's functional behavior is converted to machine code via translation to an underlying conventional language such as Ada. The PSDL program's temporal behavior is converted to machine code via a scheduling algorithm which is also implemented in a conventional language. These two programs are then compiled and executed together to achieve the

real-time characteristics desired in PSDL. The research in this thesis shows that the translation part of the extended compiler technique is feasible and nearly complete. When the translator is combined with the results of research on the static and dynamic schedulers, the execution support system of the CAPS will become complete and automated computer aided prototyping will be significantly propelled toward reality.

APPENDIX A. ADA IMPLEMENTATIONS OF PSDL CONSTRUCTS

A. *GENERIC SAMPLED DATA STREAM PACKAGE*

```
with PSDL_SYSTEM;

generic
 type ELEMENT_TYPE is private;

package SAMPLED_STREAM is
    task DATA_STREAM  is
        pragma PRIORITY (5);
        entry CHECK (NEW_DATA : out BOOLEAN);
        entry GET   (VALUE : out ELEMENT_TYPE);
        entry PUT   (VALUE : in  ELEMENT_TYPE);
    end DATA_STREAM;
    function FRESH return BOOLEAN;
end SAMPLED_STREAM;

package body SAMPLED_STREAM is

    type DATA_STREAM_TOKEN  is
        record
            INITIALIZED,
            NEW_DATA    : BOOLEAN := false;
            VALUE   :   ELEMENT_TYPE;
        end record;

    task body DATA_STREAM  is

     BUFFER : DATA_STREAM_TOKEN;

    begin
        loop
        select
            accept CHECK (NEW_DATA : out BOOLEAN) do
                NEW_DATA := BUFFER.NEW_DATA;
            end CHECK;
        or
```

```
            accept GET (VALUE : out ELEMENT_TYPE) do
                if not BUFFER.INITIALIZED then
                    raise PSDL_SYSTEM.BUFFER_UNDERFLOW;
                else
                    VALUE := BUFFER.N_VALUE;
                    BUFFER.NEW_DATA := false;
                end if;
            end GET;

        or
            accept PUT (INVALUE : in ELEMENT_TYPE) do
                BUFFER .VALUE := INVALUE;
                BUFFER.NEW_DATA := true;
                BUFFER.INITIALIZED := true;
            end PUT;

        or
            terminate;

        end select;
        end loop;
    end DATA_STREAM;

    function FRESH return BOOLEAN is
        RESULT : BOOLEAN;
    begin
        CHECK(RESULT);
        return RESULT;
    end FRESH;
end SAMPLED_STREAM;
```

B. GENERIC DATA FLOW DATA STREAM PACKAGE

```
with PSDL_SYSTEM;

generic
type ELEMENT_TYPE is private;

package DATAFLOW_STREAM is
```

```
    task DATA_STREAM is
        pragma PRIORITY (5);
        entry CHECK (NEW_DATA : out BOOLEAN);
        entry GET   (VALUE : out ELEMENT_TYPE);
        entry PUT   (VALUE : in  ELEMENT_TYPE);
    end DATA_STREAM;

    function FRESH return BOOLEAN;
end DATAFLOW_STREAM;

package body DATAFLOW_STREAM is

    type DATA_STREAM_TOKEN is
        record
            INITIALIZED,
            NEW_DATA    : BOOLEAN := false;
            VALUE   :  ELEMENT_TYPE;
        end record;

    task body DATA_STREAM is

        BUFFER : DATA_STREAM_TOKEN;

    begin
        loop
        select
            accept CHECK (NEW_DATA : out BOOLEAN) do
                NEW_DATA := BUFFER.NEW_DATA;
            end CHECK;

        or
            accept GET (OUTVALUE : out ELEMENT_TYPE) do
                if not (BUFFER.INITIALIZED and BUFFER.NEW_DATA) then
                    raise PSDL_SYSTEM.BUFFER_UNDERFLOW;
                else
                    OUTVALUE := BUFFER.VALUE;
                     BUFFER.NEW_DATA := false;
                end if;
            end GET;

        or
            accept PUT (INVALUE : in ELEMENT_TYPE) do
                if BUFFER.NEW_DATA then
```

```
                raise PSDL_SYSTEM.BUFFER_OVERFLOW;
            else
                BUFFER.VALUE := INVALUE;
                BUFFER.NEW_DATA := true;
                BUFFER.INITIALIZED := true;
            end if;
        end PUT;

    or
        terminate;
    end select;
    end loop;
end DATA_STREAM;


function FRESH return BOOLEAN is
    RESULT : BOOLEAN;
begin
    CHECK(RESULT);
    return(RESULT);
end FRESH;

end DATAFLOW_STREAM;
```

C. GENERIC SAMPLED STATE VARIABLE PACKAGE

```
with PSDL_SYSTEM;

generic
type ELEMENT_TYPE is private;
     INITIAL_VALUE : ELEMENT_TYPE;

package SAMPLED_STATE_VAR is

    task DATA_STREAM is
        pragma PRIORITY (5);
        entry CHECK (NEW_DATA : out BOOLEAN);
        entry GET  (VALUE : out ELEMENT_TYPE);
        entry PUT  (VALUE : in ELEMENT_TYPE);
    end DATA_STREAM;
    function FRESH return BOOLEAN;
```

```
end SAMPLED_STATE_VAR;

package body SAMPLED_STATE_VAR is
   type DATA_STREAM_TOKEN is
      record
         INITIALIZED,
         NEW_DATA   : BOOLEAN := false;
         VALUE  :  ELEMENT_TYPE := INITIAL_VALUE;
      end record;

   task body DATA_STREAM is

      BUFFER : DATA_STREAM_TOKEN;

   begin
      loop
      select
         accept CHECK (NEW_DATA : out BOOLEAN) do
            NEW_DATA := BUFFER.NEW_DATA;
         end CHECK;

      or
         accept GET (OUTVALUE : out ELEMENT_TYPE) do
            if not BUFFER.INITIALIZED then
               raise PSDL_SYSTEM.BUFFER_UNDERFLOW;
            else
               OUTVALUE := BUFFER.VALUE;
               BUFFER.NEW_DATA := false;
            end if;
         end GET;

      or
         accept PUT (INVALUE : in ELEMENT_TYPE) do
            BUFFER.VALUE  :=  INVALUE;
            BUFFER. INITIALIZED := true,
            BUFFER.NEW_DATA   :=  true,
         end PUT;

      or
         terminate;
      end select;
      end loop;
   end DATA_STREAM;
```

```
    function FRESH return BOOLEAN is
        RESULT : BOOLEAN;
    begin
        CHECK(RESULT);
        return RESULT;
    end FRESH;

end SAMPLED_STATE_VAR;
```

D. GENERIC DATA FLOW STATE VARIABLE PACKAGE

```
with PSDL_SYSTEM;

generic
type ELEMENT_TYPE is private;
     INITIAL_VALUE : ELEMENT_TYPE;

package DATAFLOW_STATE_VAR is

    task DATA_STREAM is
        pragma PRIORITY (5);
        entry CHECK (NEW_DATA : out BOOLEAN);
        entry GET   (VALUE : out ELEMENT_TYPE);
        entry PUT   (VALUE : in  ELEMENT_TYPE);
    end DATA_STREAM;

    function FRESH return BOOLEAN;
end DATAFLOW_STATE_VAR;


package body DATAFLOW_STATE_VAR is

    type DATA_STREAM_TOKEN is
        record
            INITIALIZED,
            NEW_DATA   : BOOLEAN := false;
            VALUE  : ELEMENT_TYPE := INITIAL_VALUE;
        end record;

    task body DATA_STREAM is
```

```
        BUFFER : DATA_STREAM_TOKEN;
    begin
        loop
        select
            accept CHECK (NEW_DATA : out BOOLEAN) do
                NEW_DATA := BUFFER.NEW_DATA;
            end CHECK;

        or
            accept GET (OUTVALUE : out ELEMENT_TYPE) do
                if not (BUFFER.INITIALIZED and BUFFER.NEW_DATA) then
                    raise PSDL_SYSTEM.BUFFER_UNDERFLOW;
                else
                    OUTVALUE := BUFFER.VALUE;
                    BUFFER.NEW_DATA := false;
                end if;
            end GET;

        or
            accept PUT (INVALUE : in ELEMENT_TYPE) do
                if BUFFER.NEW_DATA then
                    raise PSDL_SYSTEM.BUFFER_OVERFLOW;
                else
                    BUFFER.VALUE := INVALUE;
                    BUFFER.NEW_DATA := true;
                    BUFFER.INITIALIZED := true;
                end if;
            end PUT;

        or
            terminate;
        end select;
        end loop;
    end DATA_STREAM;

    function FRESH return BOOLEAN is
        RESULT : BOOLEAN;
    begin
        CHECK(RESULT);
        return(RESULT);
    end FRESH;

end DATAFLOW_STATE_VAR;
```

E. GENERIC TIMER PACKAGE

```
with PSDL_SYSTEM;
package PSDL_TIMER is
  subtype MILLISEC is new PSDL_SYSTEM.TIMER;

  procedure RESET ;
  procedure START ;
  procedure STOP ;
  function  READ return MILLISEC;

end PSDL_TIMER;


with CALENDAR;
use CALENDAR;
package body PSDL_TIMER is

 type STATE is (INITIAL, RUNNING, STOPPED);
 type TIMER is
  record
    START_TIME,
    STOP_TIME   : CALENDAR.TIME;
    ELAPSED_TIME : DURATION;
    PRESENT_STATE : STATE;
  end record;

   WATCH : TIMER;

 function READ return MILLISEC is
   CONVERSION_FACTOR : constant DURATION := 1000.0; -- Converts to
 begin                                              -- milliseconds
  case PRESENT_STATE is

     when RUNNING => return MILLISEC((CLOCK
                          - WATCH.START_TIME
                          + WATCH.ELAPSED_TIME)
                           * CONVERSION_FACTOR);
     when others => return MILLISEC(WATCH.ELAPSED_TIME);
  end case;
end READ;
```

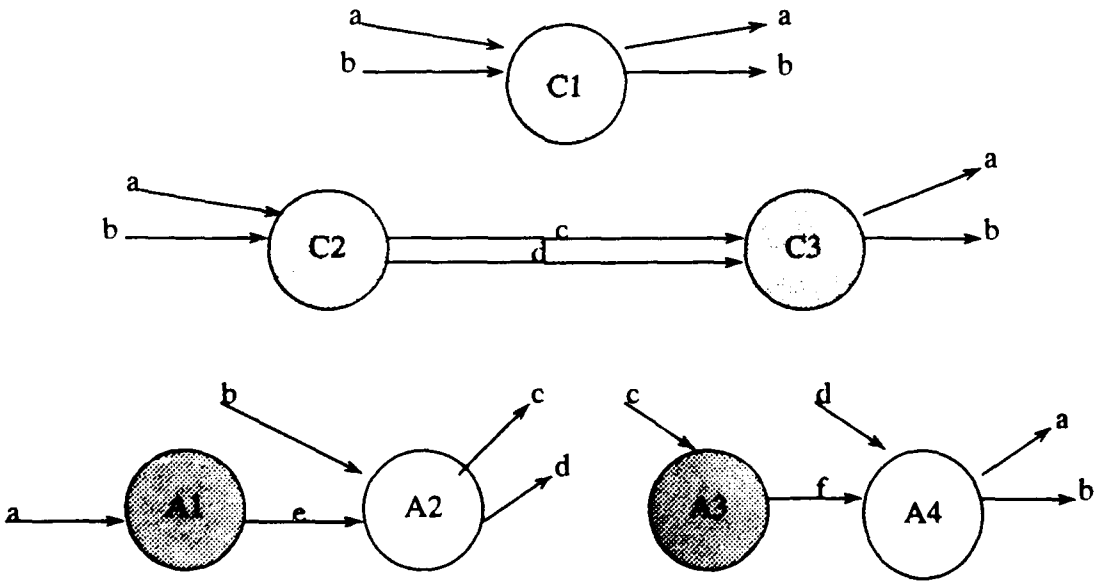
```
  procedure RESET  is

    begin
      case WATCH.PRESENT_STATE is
        when STOPPED =>  WATCH.ELAPSED_TIME := 0.0;
                WATCH.PRESENT_STATE := INITIAL;
        when others =>  null;
      end case;
    end RESET;


procedure START is

begin
 case WATCH.PRESENT_STATE is
   when RUNNING    =>  null;
   when others     =>  WATCH.START_TIME := CALENDAR.CLOCK;
                       WATCH.PRESENT_STATE := RUNNING;
 end case;
end START;


procedure STOP  is

begin
 case WATCH.PRESENT_STATE is
   when RUNNING    =>   WATCH.ELAPSED_TIME := CALENDAR.CLOCK
                - WATCH.START_TIME
                + WATCH.ELAPSED_TIME;
                WATCH.PRESENT_STATE := STOPPED;

   when others     =>  null;
 end case;
end STOP;

end PSDL_TIMER;
```

APPENDIX B. SPECIFICATION PACKAGE TRANSLATION EXAMPLE

The purpose of this example is to demonstrate the translation of a composite operator into a set of specification packages. The PSDL rules of visibility state that data streams declared in a composite operator are visible to the composite operator's subcomponents. The difficulty to overcome in this thesis was finding a way to allow this sort of hierarchical visibility structure in a flat physical structure.

This example takes a simple PSDL operator and decomposes it into its subcomponents. A data flow diagram is presented to show the hierarchical structure of a PSDL operator and the data streams at each operator. The PSDL representation of the data flow diagram is then shown. Finally, the PSDL program is translated into a set of specification packages which demonstrates the flat structure of the Ada representation and how the lines of visibility can be established using the Ada **with** and **renames** clauses.

A. DATA FLOW DIAGRAM



B. PSDL REPRESENTATION OF DATA FLOW DIAGRAM

```
OPERATOR C1
SPECIFICATION
   STATES a : INTEGER INITIALLY 0,
              b : FLOAT  INITIALLY 1.0;
END

IMPLEMENTATION GRAPH
   a.EXTERNAL -> C2
   b.EXTERNAL -> C2
   a.C3 -> EXTERNAL
   b.C3 -> EXTERNAL
   c.C2 -> C3
   d.C2 -> C3

   DATA STREAM c, d : INTEGER;
   CONTROL CONSTRAINTS
     OPERATOR C2
       PERIOD 200
     OPERATOR C3
       PERIOD 200
   DESCRIPTION
   { This is the root operator. It is not allowed to have any external input or output
   data streams. To get information into the system, state variables are used. Note
   that the subcomponents are composite operators also, thus they may not have
   any control constraints like triggers, outputs, timers, or exceptions. }
END

OPERATOR C2
SPECIFICATION
   INPUT a : INTEGER,
         b : FLOAT
   OUTPUT c, d : INTEGER
END

IMPLEMENTATION GRAPH
   a.EXTERNAL -> A1
```

```
    b.EXTERNAL -> A2
    c.A2 -> EXTERNAL
    d.A2 -> EXTERNAL
    e.A1 -> A2

    DATA STREAM e : FLOAT

    CONTROL CONSTRAINTS
        OPERATOR A1
            TRIGGERED IF a >= 0
        OPERATOR A2
            TRIGGERED BY SOME b
            OUTPUT c IF c > 0
    DESCRIPTION
    { Operator C2 is a composite operator with atomic subcomponents A1 and A2.
    Operator C3 requires visibility to all of its parent's data streams, a, b, c, and d. }
END


OPERATOR C3
SPECIFICATION
    INPUT c, d : INTEGER
    OUTPUT a : INTEGER,
           b : FLOAT
END

IMPLEMENTATION GRAPH
    c.EXTERNAL -> A3
    d.EXTERNAL -> A4
    a.A4 -> EXTERNAL
    b.A4 -> EXTERNAL
    f.A3 -> A4

    DATA STREAM f : FLOAT


    CONTROL CONSTRAINTS
        OPERATOR A3
```

```
        TRIGGERED
      OPERATOR A4
        TRIGGERED
   DESCRIPTION
   { Operator C2 is a composite operator with atomic subcomponents A3 and A4.
   Operator C3 requires visibility to all of its parent's data streams, a, b, c, and d. }
END

OPERATOR A1
SPECIFICATION
   INPUT a : INTEGER
   OUTPUT e : FLOAT
END

IMPLEMENTATION ADA A1 { The atomic operator A1 requires visibility to data
   streams a and e in C2. }
END

OPERATOR A2
SPECIFICATION
   INPUT b, e : FLOAT
   OUTPUT c, d : INTEGER
END

IMPLEMENTATION ADA A2 { The atomic operator A2 requires visibility to data
   streams b, c, d, and e in C2 }
END

OPERATOR A3
SPECIFICATION
   INPUT c : INTEGER
   OUTPUT f : FLOAT
END

IMPLEMENTATION ADA A3 {The atomic operator A3 requires visibility to data
   streams c and f in C3}
```

```
OPERATOR A4
SPECIFICATION
   INPUT d : INTEGER,
         f : FLOAT
   OUTPUT a : INTEGER
          b : FLOAT
END

IMPLEMENTATION ADA A4 { The atomic operator A4 requires visibility to data
   streams a, b, d, and f in C3. }
END
```

C. ADA SPECIFICATION PACKAGES FOR OPERATORS C1, C2, C3, A1, AND A3

```
package C1_SPEC is
-- This is the specification package for the root operator. It instantiates data
-- streams a, b, c, and d.

   package DSa is new SAMPLED_STATE_VAR(INTEGER,0);
   package DSb is new SAMPLED_STATE_VAR(FLOAT,1.0);
   package DSc is new SAMPLED_STREAM(INTEGER);
   package DSd is new SAMPLED_STREAM(FLOAT);
end C1_SPEC;

with C1_SPEC;
package C2_SPEC is
-- This is the specification package for composite operator C2. It establishes a link
-- with its parent, C1, via the statement with C1_SPEC. Visibility is required to
-- data streams a, b, c, and d.

   package DSa renames C1_SPEC.DSa;
   package DSb renames C1_SPEC.DSb;
   package DSc renames C1_SPEC.DSc;
   package DSd renames C1_SPEC.DSd;
   package DSe is new SAMPLED_STREAM(FLOAT);
end C2_SPEC;
```

```
with C1_SPEC;
package C3_SPEC is
-- This is the specification package for operator C3.  It requires visibility to data
-- streams a, b, c, and d in C1_SPEC.

    package DSa renames C1_SPEC.DSa;
    package DSb renames C1_SPEC.DSb;
    package DSc renames C1_SPEC.DSc;
    package DSd renames C1_SPEC.DSd;
    package DSf is new SAMPLED_STREAM(FLOAT);
end C3-SPEC;

    with C2_SPEC; use C2_SPEC;
    procedure C2_A1 is
        TL_a  :  INTEGER;
        TL_e  :  FLOAT;
    begin
        loop
        if true then
            DSa.GET(TL_a);
            if TL_a >= 0 then
                A1(TL_a, TL_e);
                DSe.PUT(TL_e);
            end if;
        end if;
        exit;
        end loop;
    end C2_A1;

    with C3_SPEC; use C3_SPEC;
    procedure C3_A3 is
        TL_c  :  INTEGER;
        TL_f  :  FLOAT;
    begin
        loop
        if true then
            DSc.GET(TL_c);
            if true then
```

```
                A3(TL_c,TL_f);
                DSf.PUT(TL_f);
            end if;
        end if;
        exit;
        end loop;
    end C3_A3;
```

APPENDIX C. PSDL GRAMMAR

The conventions used for symbology in this grammar are standard. {Curly braces} indicate item which may appear zero or more times. [Square brackets] indicate items which may appear zero or one time in a rule. Bold face items are terminal keyword symbols in the grammar. "Double quotes" indicate character literals in the grammar. The "|" vertical bar indicates a list of options from which no more than one item may be selected. This grammar represents the updated version of the PSDL grammar as of 26 Oct 1988.

Start = psdl

psdl = { component }

component = data_type | operator

data_type = **type** id type_spec type_impl

operator = **operator** id operator_spec operator_impl

type_spec = **specification** [type_decl] {**operator** id operator_spec}
[functionality] **end**

type_impl = **implementation ada** id "{" text "}"
| **implementation** type_name {**operator** id operator_impl} **end**

operator_spec = **specification** {interface} [functionality] **end**

operator_impl = **implementation ada** id "{" text "}"
| **implementation** psdl_impl

type_decl = id_list ":" type_name {"," id_list ":" type_name}

functionality = [keywords] [informal_desc] [formal_desc]

psdl_impl = data_flow_diagram [streams] [timers] [control_constraints]
[informal_desc] **end**

type_name = id "[" type_decl "]"
    | *id*

interface = attribute {reqmts_trace}

id_list = id {"," id }

keywords = **keywords** id_list

informal_desc = **description** "{" text "}"

formal_desc = **axioms** "{" text "}"

data_flow_diagram = **graph** {link}

streams = **data stream** type_decl

timers = **timer** id_list

attribute = generic_param
    | input
    | output
    | states
    | *exceptions*
    | timing_info

generic_param = **generic** type_decl

input = **input** type_decl

output = **output** type_decl

states = **states** type_decl **initially** expression_list

exceptions = **exceptions** id_list

timing_info = [**maximum execution time** time]
    [**minimum calling period** time]
    [**maximum response time** time]
reqmts_trace = **by requirements** id_list
link = id "." id [":" time] "->" id

control_constraints = **control constraints** {constraint}

```
constraint = operator id
    [triggered [trigger] [if predicate] [reqmts_trace]]
    [period time [reqmts_trace]]
    [finish within time [reqmts_trace]]
    {constraint_options}

trigger = by all id_list
    | by some id_list

constraint_options = output id_list if predicate [reqmts_trace]
    | exception id [if predicate] [reqmts_trace]
    | timer_op id [if predicate] [reqmts_trace]

timer_op = read timer
    | reset timer
    | start timer
    | stop timer

expression_list = expression {"," expression}

time = integer [unit]

unit = ms | sec | min | hours

expression = constant
    | id
    | type_name "." id "(" expression_list ")"

predicate = relation {bool_op relation}

relation = simple_expression
    | simple_expression rel_op simple_expression

simple_expression = [sign] integer [unit]
    | [sign] real
    | [not] id
    | string
    | [not] "(" predicate ")"
    | [not] boolean_constant
bool_op = and | or

rel_op = "<" | "<=" | ">" | ">=" | "=" | "/=" | ":"
```

real = integer "." integer

integer = digit {digit}

boolean_constant = **true** | **false**

numeric_constant = real | integer

constant = numeric_constant | boolean_constant

sign = "+" | "-"

char = any printable character except "}"

digit = "0 .. 9"

letter = "a .. z" | "A .. Z" | "_"

alpha_numeric = letter | digit

id = letter {alpha_numeric}

string = """ {char} """

text = {char}

APPENDIX D. KODIYAK PROGRAM LISTING

The following is a listing of the Kodiyak program used to generate the translator in this thesis. It is based upon the Kodiyak program produced by Moffitt in [Ref. 17]. This program is capable of producing a translator which will translate a subset of the PSDL grammar. The constructs not translated by this version include:

- PSDL exceptions.
- User-defined types.

```
!definitions of lexical classes

%define  Digit          :[0-9]
%define  Int            :{Digit}+
%define  Letter         :[a-zA-Z_]
%define  Alpha          :({Letter}|{Digit})
%define  Blank          :[ \t\n]
%define  Char           :[^{}]
%define Quote           :["]

! definitions of white space

  :{Blank}+

! definitions of compound symbols and keywords

GTE                     :">="
LTE                     :"<="
NEQV                    :"/="
ARROW                   :"->"
TYPE                    :type|TYPE
OPERATOR                :operator|OPERATOR
SPECIFICATION           :specification|SPECIFICATION
END                     :end|END
GENERIC                 :generic|GENERIC
INPUT                   :input|INPUT
OUTPUT                  :output|OUTPUT
STATES                  :states|STATES
```

```
INITIALLY           :initially|INITIALLY
EXCEPTIONS          :exceptions|EXCEPTIONS
NORMAL              :normal|NORMAL
MAX_EXEC_TIME
  :maximum[ ]execution[ ]time|MAXIMUM[ ]EXECUTION[ ]TIME
MAX_RESP_TIME
  :maximum[ ]response[ ]time|MAXIMUM[ ]RESPONSE[ ]TIME
MIN_CALL_PERIOD
  :minimum[ ]calling[ ]period|MINIMUM[ ]CALLING[ ]PERIOD
MS                  :ms|MS
SEC                 :sec|SEC
MIN                 :min|MIN
HOURS               :hours|HOURS
BY                  :by[ ]requirements|BY[ ]REQUIREMENTS
KEYWORDS            :keywords|KEYWORDS
DESCRIPTION         :description|DESCRIPTION
AXIOMS              :axioms|AXIOMS
IMPLEMENTATION      :implementation|IMPLEMENTATION
ADA                 :ada|Ada|ADA
GRAPH               :graph|GRAPH
DATA_STREAM         :data[ ]stream|DATA[ ]STREAM
TIMER               :timer|TIMER
CONTROL             :control[ ]constraints|CONTROL[ ]CONSTRAINTS
TRIGGERED           :triggered|TRIGGERED
ALL                 :by[ ]all|BY[ ]ALL
SOME                :by[ ]some|BY[ ]SOME
PERIOD              :period|PERIOD
FINISH              :finish[ ]within|FINISH[ ]WITHIN
EXCEPTION           :exception|EXCEPTION
READ                :read[ ]timer|READ[ ]TIMER
RESET               :reset[ ]timer|RESET[ ]TIMER
START               :start[ ]timer|START[ ]TIMER
STOP                :stop[ ]timer|STOP[ ]TIMER
IF                  :if|IF
NOT                 :"~" | "not" | "NOT"
AND                 :"&" | "and" | "AND"
OR                  :"|" | "or" | "OR"
TRUE                :true|TRUE
```

```
FALSE                   :false|FALSE
ID                      :{Letter}{Alpha}*
STRING_LITERAL          :{Quote}{Char}*{Quote}
INTEGER_LITERAL         :{Int}
REAL_LITERAL            :{Int}"."{Int}
TEXT                    :"{"{Char}*"}"


! operator precedences
! %left means group and evaluate from the left

%left  OR;
%left  AND;
%left  NOT;
%left  '<', '>', '=', GTE, LTE, NEQV;
%left  ':';

%%
! attribute declarations for nonterminal symbols

start { trn: string; };
psdl { trn: string;
     uncond_output_map:string->string;
     out_env:string->string;
     in_env:string->string; };
component { trn: string;
          uncond_output_map_in:string->string;
          uncond_output_map_out:string->string;
          in_env:string->string;
          out_env:string->string; };

data_type { trn: string;
          in_env:string->string; };
operator { trn: string;
         uncond_output_map_in:string->string;
         uncond_output_map_out:string->string;
         in_env:string->string;
         out_env:string->string; };
```

```
        type_spec { trn: string;
          in_env:string->string; };
    type_decl_1_list { trn: string;
              in_env:string->string; };
    type_decl { trn: string;
          in_env,out_env :string->string;
          opid:string;
          action_code:string;
          ucond_output:string; };

    op_spec_0_list { trn: string;
              in_env:string->string; };

    operator_spec { opid:string;
              ds_decl:string;
              state_decl:string;
              ucond_output:string;
              excp_decl:string;
              in_env,out_env :string->string; }

    interface { in_env,out_env :string->string;
          in_parm, out_parm : string;
          ds_decl: string;
          state_decl:string;
          excp_decl:string;
          ucond_output:string;
          opid:string; };

    attribute { ds_decl: string;
          in_env,out_env :string->string;
          in_parm, out_parm: string;
          opid:string;
          state_decl:string;
          ucond_output:string;
          excp_decl:string; };

    time { trn: string; };
    unit { value: int; };
```

```
id_list { trn: string;
        action_code:string;
        tname:string;
        opid:string;
        ucond_output:string;
        count : int;
        exp_env:int->string;
        in_env:string->string;
        out_env:string->string; };

reqmts_trace { trn: string; };
functionality { trn: string; };
keywords { trn: string; };
informal_desc { trn: string; };
formal_desc { trn: string; };
type_impl { trn: string; };
op_impl_0_list { trn: string; };
operator_impl { trn: string;
              out_env:string->string;
              in_env:string->string;
              uncond_output_map:string->string;
              loc_ds_decl:string;
              timer_decl:string;
              opid:string; };
psdl_impl { trn: string;
          parent : string;
          uncond_output_map:string->string;
          in_env:string->string;
          out_env:string->string;
          loc_ds_decl:string;
          timer_decl:string; };

data_flow_diagram { trn: string;
                  in_env, decl_map : string-> string; };

link_0_list { trn: string;
            in_env,in_decls,out_decls : string->string; };
```

```
link { trn: string;
    in_env,in_decls,out_decls : string->string;  };

opt_time { trn: string; };
type_name { trn: string; };
timers { trn: string; };
control_constraints { trn: string;
                parent : string;
                uncond_output_map:string->string;
                in_env:string->string;
                out_env:string->string;
                decl_map : string->string;
               };
constraint_options { trn : string;
                in_env: string->string;
                out_env:string->string;
                opid:string;
               };
more_constraints {trn : string;
               parent:string;
               uncond_output_map:string->string;
               in_env:string->string;
               out_env:string->string;
               decl_map : string->string;
              };
opt_trig { out_env:string->string;
        in_env: string->string;
        streams_check:string;
        end_if_streams:string;
        pred:string;
        end_if_pred:string;
       };

trigger {if: string;
      end_if:string;
      in_env, out_env:string->string; };
opt_per { trn: string; };
opt_fin_w { trn: string; };
```

```
streams {trn: string;
      in_env,out_env :string->string; };
timer_op { trn: string; };
opt_if_predicate { if: string;
              end_if:string;
              parent:string;
              in_env : string-> string;  };
predicate { trn: string;
         in_env: string->string;
         type: string; };
expression_list { trn: string;
             count:int;
             exp_env:int->string; };
expression { trn: string; };
relation {trn: string;
       in_env: string->string;
       type: string; };
simple_expression { trn: string;
              parent: string;
              in_env: string->string;
              type:string; };
rel_op {trn: string;
     left_op:string;
     right_op: string;
     opn_type: string;
     parent: string; };
sign {trn: string; };

!attrbute declarations for terminal symbols

ID{ %text: string; };
TEXT{ %text: string; };
STRING_LITERAL{ %text: string; };
INTEGER_LITERAL{ %text: string; };
REAL_LITERAL {%text: string; };

%%
!psdl grammar
```

```
start
 : psdl
  { %output(["with PSDL_SYSTEM;\nuse PSDL_SYSTEM;\npackage TL is\n",
              psdl.trn,"end TL;\n"]);
       psdl.in_env =  psdl.out_env;
      }
 ;

psdl
 : component psdl
  { psdl[1].trn = [component.trn,"\n",psdl[2].trn];
       psdl[1].out_env = component.out_env +| psdl[2].out_env;
       psdl[1].uncond_output_map = component.uncond_output_map_out
         +| psdl[2].uncond_output_map;
       component.in_env = psdl[1].in_env;
       component.uncond_output_map_in = psdl[2].uncond_output_map;
       psdl[2].in_env = psdl[1].in_env ;
      }
 |
  { psdl.trn = "";
       psdl.out_env = {(?:string:"")};
       psdl.uncond_output_map = {(?:string:"")} ;
      }
 ;

component
 : data_type
  { component.trn = "";
       component.out_env = {(?:string:"")};
       component.uncond_output_map_out = {(?:string:"")};
       data_type.in_env = component.in_env;
      }

 | operator
  { component.trn = operator.trn;
       component.out_env = operator.out_env;
       component.uncond_output_map_out = operator.uncond_output_map_out;
       operator.in_env = component.in_env;
```

```
        operator.uncond_output_map_in = component.uncond_output_map_in;
       }
 ;

data_type
 : TYPE ID type_spec type_impl
  { data_type.trn = "";
       type_spec.in_env = data_type.in_env;
       }
 ;

operator
 : OPERATOR  ID operator_spec operator_impl
  { operator.trn =
       (operator.in_env(ID.%text^"CONSTRUCT") == "composite_operator"
        -> ["\npackage ",
          ID.%text,"_SPEC is\n",operator_spec.ds_decl,"\n",
          operator_impl.loc_ds_decl,"\n",
          operator_spec.state_decl,"\n",
          operator_impl.timer_decl,"\n",
          operator_spec.excp_decl,"\nend ",
          ID.%text,"_SPEC;\n",operator_impl.trn]
        #  ""
       );
       operator.uncond_output_map_out =
         {(ID.%text:operator_spec.ucond_output)};

       operator_spec.opid = ID.%text;
       operator_spec.in_env = operator.in_env;
       operator_impl.opid = ID.%text;
       operator_impl.in_env = {("PARENT":ID.%text)} +| operator.in_env;
       operator_impl.uncond_output_map = operator.uncond_output_map_in;
       operator.out_env = operator_spec.out_env +| operator_impl.out_env;
       }
 ;

type_spec
 : SPECIFICATION type_decl_1_list op_spec_0_list functionality END
```

```
  { type_spec.trn = "";
      type_decl_1_list.in_env = type_spec.in_env;
      op_spec_0_list.in_env = type_spec.in_env;
    }
 ;

type_decl_1_list
 : type_decl
  { type_decl_1_list.trn = type_decl.trn;
      type_decl.action_code = "type";
      type_decl.in_env = type_decl_1_list.in_env;
    }

 |
  {type_decl_1_list.trn = "";}
 ;

type_decl
 : id_list ':' type_name
  { type_decl.trn = id_list.trn;
      type_decl.out_env = id_list.out_env;
      type_decl.ucond_output = id_list.ucond_output;
      id_list.in_env = type_decl.in_env;
      id_list.action_code = type_decl.action_code;
      id_list.tname = type_name.trn;
      id_list.opid = type_decl.opid;
      id_list.count = 1;
      id_list.exp_env = {(?:int:"")};
    }

   | id_list ':' type_name ',' type_decl
    { type_decl[1].trn = id_list.trn ^ type_decl[2].trn;
      type_decl[1].out_env = id_list.out_env +| type_decl[2].out_env;
      type_decl.ucond_output = id_list.ucond_output
                    ^ type_decl[2].ucond_output;

      id_list.in_env = type_decl[1].in_env;
      id_list.action_code = type_decl[1].action_code;
```

```
        id_list.tname = type_name.trn;
        id_list.opid = type_decl[1].opid;
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
        type_decl[2].in_env = type_decl[1].in_env;
        type_decl[2].opid = type_decl[1].opid;
        type_decl[2].action_code = type_decl[1].action_code;
       }
 ;
op_spec_0_list
 : op_spec_0_list OPERATOR ID operator_spec
  { op_spec_0_list[1].trn = "";
        operator_spec.in_env = op_spec_0_list.in_env;
        op_spec_0_list[2].in_env = op_spec_0_list[1].in_env;
       }
 |
  { op_spec_0_list.trn = ""; }
 ;

operator_spec
 : SPECIFICATION interface functionality END
       { operator_spec.ds_decl = interface.ds_decl;
        operator_spec.state_decl = interface.state_decl;
        operator_spec.excp_decl = interface.excp_decl;
        operator_spec.ucond_output = interface.ucond_output;
        operator_spec.out_env =
         (interface.out_env(operator_spec.opid^"INPARM") == "" ||
         interface.out_env(operator_spec.opid^"OUTPARM") == ""
         -> {((operator_spec.opid^"PROCCALL"):
            [interface.out_env(operator_spec.opid^"INPARM"),
             interface.out_env(operator_spec.opid^"OUTPARM")])}
         # {((operator_spec.opid^"PROCCALL"):
            [interface.out_env(operator_spec.opid^"INPARM"),",",
             interface.out_env(operator_spec.opid^"OUTPARM")])}
        ) +| interface.out_env;

        interface.in_env = operator_spec.in_env;
        interface.opid = operator_spec.opid;
```

```
        }
    ;

interface
  : interface attribute reqmts_trace
    { interface[1].ds_decl =
          [interface[2].ds_decl,"\n",attribute.ds_decl];
     interface[1].state_decl =
          [interface[2].state_decl,"\n",attribute.state_decl];
     interface[1].excp_decl =
          [interface[2].excp_decl,"\n",attribute.excp_decl];
        interface[1].in_parm = interface[2].in_parm ^ attribute.in_parm;
        interface[1].out_parm = interface[2].out_parm ^ attribute.out_parm;

        interface[1].out_env =
          {((interface[1].opid^"INPARM"):interface[1].in_parm)
           ((interface[1].opid^"OUTPARM"):interface[1].out_parm) }
          +| interface[2].out_env  +| attribute.out_env;

        interface[1].ucond_output = interface[2].ucond_output
                          ^ attribute.ucond_output;

        interface[2].opid = interface[1].opid;
        interface[2].in_env = interface[1].in_env;
        attribute.in_env = interface[1].in_env;
        attribute.opid = interface[1].opid;
       }

  |
    {interface.ds_decl = "";
        interface.state_decl = "";
        interface.excp_decl = "";
        interface.in_parm = "";
        interface.out_parm = "";
        interface.out_env = {(?:string:"")};
        interface.ucond_output = "";
       }
  ;
```

```
attribute
 : GENERIC type_decl
       { type_decl.action_code = "";
         type_decl.opid = attribute.opid;
         type_decl.in_env = attribute.in_env;
         attribute.out_env = type_decl.out_env;
         attribute.ds_decl = "";
         attribute.state_decl = "";
         attribute.excp_decl = "";
         attribute.in_parm = "";
         attribute.out_parm = "";
         attribute.ucond_output = "";
       }

 | INPUT type_decl
       { type_decl.action_code = "input";
         type_decl.opid = attribute.opid;
         type_decl.in_env = attribute.in_env;
         attribute.out_env = type_decl.out_env;
         attribute.ds_decl = type_decl.trn;
         attribute.state_decl = "";
         attribute.excp_decl = "";
         attribute.in_parm = type_decl.out_env(attribute.opid^"INPARM");
         attribute.out_parm = "";
         attribute.ucond_output = "";

       }

 | OUTPUT type_decl
       { type_decl.action_code = "output";
         type_decl.opid = attribute.opid;
         type_decl.in_env = attribute.in_env;
         attribute.out_env = type_decl.out_env;
         attribute.ds_decl = type_decl.trn;
         attribute.state_decl = "";
         attribute.excp_decl = "";
         attribute.in_parm = "";
         attribute.out_parm = type_decl.out_env(attribute.opid^"OUTPARM");
```

```
        attribute.ucond_output = type_decl.ucond_output;

      }

| STATES id_list ':' ID INITIALLY expression_list
      { id_list.action_code = "states";
        id_list.opid = attribute.opid;
        id_list.tname = ID.%text;
        id_list.count = 1;
        id_list.exp_env = expression_list.exp_env;
        id_list.in_env = attribute.in_env;
        expression_list.count = 1;
        attribute.out_env = id_list.out_env;
        attribute.ds_decl = "";
        attribute.state_decl = id_list.trn;
        attribute.excp_decl = "";
        attribute.in_parm = "";
        attribute.out_parm = "";
        attribute.ucond_output = "";

      }

| EXCEPTIONS id_list
      { id_list.action_code = "excp";
        id_list.tname = "exception";
        id_list.opid = attribute.opid;
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
        id_list.in_env = attribute.in_env;
        attribute.out_env = id_list.out_env;
        attribute.ds_decl = "";
        attribute.state_decl = "";
        attribute.excp_decl = [id_list.trn," : PSDL_EXCEPTION;\n"];
        attribute.in_parm = "";
        attribute.out_parm = "";
        attribute.ucond_output = "";

      }
```

```
| MAX_EXEC_TIME time
      { attribute.ds_decl = "";
        attribute.state_decl = "";
        attribute.excp_decl = "";
        attribute.out_env = {(?:string:"")};
        attribute.in_parm = "";
        attribute.out_parm = "";
        attribute.ucond_output = "";

      }

    | MIN_CALL_PERIOD time
      { attribute.ds_decl = "";
        attribute.state_decl = "";
        attribute.excp_decl = "";
        attribute.out_env = {(?:string:"")};
        attribute.in_parm = "";
        attribute.out_parm = "";
        attribute.ucond_output = "";

      }

    | MAX_RESP_TIME time
      { attribute.ds_decl = "";
        attribute.state_decl = "";
        attribute.excp_decl = "";
        attribute.out_env = {(?:string:"")};
        attribute.in_parm = "";
        attribute.out_parm = "";
        attribute.ucond_output = "";

      }


id_list
 : ID ',' id_list
   { id_list[1].trn =
       (id_list[1].action_code == "input" ||
```

```
 id_list[1].action_code == "output"
   -> (id_list[1].in_env(id_list[1].opid^"PARENT") == ""
      -> ["package DS",ID.%text," is new ",
        (id_list[1].in_env(ID.%text^"BUFF_TYPE") == "fifo"
          -> "FIFO_BUFFER"
          # "SAMPLED_BUFFER")
        ,"(",id_list[1].tname,");\n"]
      # ["package DS",ID.%text," renames ",
        id_list[1].in_env(id_list[1].opid^"PARENT"),"_SPEC.DS",
        ID.%text,";\n"]) ^ id_list[2].trn

# id_list[1].action_code == "states"
   -> ["package DS",ID.%text," is new STATE_VARIABLE(",
      id_list[1].tname,", ",id_list[1].exp_env(id_list[1].count),
      ");\n"] ^ id_list[2].trn

#  id_list[1].action_code == "excp"
   -> ["EX",ID.%text,",",id_list[2].trn]

# id_list[1].action_code == "stream"
   -> ["package DS",ID.%text," is new ",
      (id_list[1].in_env(ID.%text^"BUFF_TYPE") == "fifo"
       -> "FIFO_BUFFER"
       # "SAMPLED_BUFFER"), "(",id_list[1].tname,");\n"]
     ^ id_list[2].trn

# id_list[1].action_code == "timer"
   -> ["TL",ID.%text,",",id_list[2].trn]


# id_list[1].action_code == "by_all"
   -> [id_list[1].opid,"_SPEC.DS",ID.%text,".NEW_DATA"," AND \n",
      id_list[2].trn]

# id_list[1].action_code == "by_some"
   -> [id_list[1].opid,"_SPEC.DS",ID.%text,".NEW_DATA"," OR \n",
      id_list[2].trn]
# id_list[1].action_code == "co_output"
```

```
   -> [id_list[1].in_env(id_list[1].opid^"PARENT"),"_SPEC.DS",
      ID.%text,".WRITE(",ID.%text,");\n",id_list[2].trn]

#  "");

id_list[2].in_env = id_list[1].in_env;
id_list[2].action_code = id_list[1].action_code;
id_list[2].opid = id_list[1].opid;
id_list[2].tname = id_list[1].tname;
id_list[2].exp_env = id_list[1].exp_env;
id_list[2].count = id_list[1].exp_env(id_list[1].count + 1) <> ""
     -> id_list[1].count + 1
     #  id_list[1].count;

id_list[1].out_env =
(id_list[1].action_code == "by_all"
 -> {((ID.%text^"BUFF_TYPE"):"fifo")} +| id_list[2].out_env

 # id_list[1].action_code == "by_some"
   -> {((ID.%text^"BUFF_TYPE"):"sampled")} +| id_list[2].out_env

 # id_list[1].action_code == "input"
   -> {((ID.%text^"TYPE"):id_list[1].tname)
      ((ID.%text^"CONSTRUCT"):"data_stream")
      ((id_list[1].opid^"INPARM"):[ID.%text,",",
        id_list[2].out_env(id_list[2].opid^"INPARM")])
     } +| id_list[2].out_env

 # id_list[1].action_code == "output"
   -> {((ID.%text^"TYPE"):id_list[1].tname)
      ((ID.%text^"CONSTRUCT"):"data_stream")
      ((id_list[1].opid^"OUTPARM"):[ID.%text,",",
        id_list[2].out_env(id_list[2].opid^"OUTPARM")])
      } +| id_list[2].out_env

 # id_list[1].action_code == "stream"
   -> {((ID.%text^"TYPE"):id_list[1].tname)
      ((ID.%text^"CONSTRUCT"):"data_stream")} +| id_list[2].out_env
```

```
        # id_list[1].action_code == "states"
          -> {((ID.%text^"TYPE"):id_list[1].tname)
             ((ID.%text^"CONSTRUCT"):"data_stream")} +| id_list[2].out_env

        # id_list[1].action_code == "excp"
          -> {((ID.%text^"CONSTRUCT"):"exception")} +| id_list[2].out_env

        # id_list[1].action_code == "timer"
          -> {((ID.%text^"CONSTRUCT"):"timer")} +| id_list[2].out_env

        # id_list[1].action_code == "co_output"
          -> {([id_list[1].opid,"_",ID.%text,"OUTPUT"]:"conditional")}
             +| id_list[2].out_env

        # {(?:string:"")}
        );

        id_list.ucond_output =
        ((id_list[1].in_env(id_list[1].opid^"_"^ID.%text^"OUTPUT")
          <> "conditional" ) &&
         (id_list[1].action_code == "output")
          -> [id_list[1].in_env(id_list[1].opid^"PARENT"),"_SPEC.DS",
              ID.%text,".WRITE(",ID.%text,");\n"]
          # ""
        ) ^ id_list[2].ucond_output;

      }


| ID
  { id_list.trn =
      (id_list.action_code == "input" ||
       id_list.action_code == "output"
        -> (id_list.in_env(id_list.opid^"PARENT") == ""
            -> ["package DS",ID.%text," is new ",
                (id_list.in_env(ID.%text^"BUFF_TYPE") == "fifo"
                 -> "FIFO_BUFFER"
                 # "SAMPLED_BUFFER")
```

```
      ,"(",id_list.tname,");\n"]
    # ["package DS",ID.%text," renames ",
      id_list.in_env(id_list.opid^"PARENT"),"_SPEC.DS",
      ID.%text,";\n"])

# id_list.action_code == "states"
    -> ["package DS",ID.%text," is new STATE_VARIABLE(",
     id_list.tname,", ",id_list.exp_env(id_list.count),
     ");\n"]
# id_list.action_code == "excp"
    -> ["EX",ID.%text]

# id_list.action_code == "stream"
  -> ["package DS",ID.%text," is new ",
     (id_list.in_env(ID.%text^"BUFF_TYPE") == "fifo"
      -> "FIFO_BUFFER"
      #  "SAMPLED_BUFFER"), "(",id_list.tname,");\n"]

# id_list.action_code == "timer"
  -> ["TL",ID.%text]

# id_list.action_code == "by_all"
  -> [id_list.opid,"_SPEC.DS",ID.%text,".NEW_DATA"]

# id_list.action_code == "by_some"
  -> [id_list.opid,"_SPEC.DS",ID.%text,".NEW_DATA"]

# id_list.action_code == "co_output"
  -> [id_list.in_env(id_list.opid^"PARENT"),"_SPEC.DS",ID.%text,
     ".WRITE(",ID.%text,");\n"]

# "");

id_list.out_env =
(id_list.action_code == "by_all"
-> {((ID.%text^"BUFF_TYPE"):"fifo")}

# id_list.action_code == "by_some"
```

```
  -> {((ID.%text^"BUFF_TYPE"):"sampled")}

 # id_list[1].action_code == "input"
  -> {((ID.%text^"TYPE"):id_list[1].tname)
     ((ID.%text^"CONSTRUCT"):"data_stream")
     ((id_list.opid^"INPARM"):ID.%text)}

 # id_list[1].action_code == "output"
  -> {((ID.%text^"TYPE"):id_list[1].tname)
     ((ID.%text^"CONSTRUCT"):"data_stream")
     ((id_list.opid^"OUTPARM"):ID.%text)}

 # id_list[1].action_code == "stream"
  -> {((ID.%text^"TYPE"):id_list[1].tname)
     ((ID.%text^"CONSTRUCT"):"data_stream")}
 # id_list[1].action_code == "states"
  -> {((ID.%text^"TYPE"):id_list[1].tname)
     ((ID.%text^"CONSTRUCT"):"data_stream")}

 # id_list.action_code == "excp"
  -> {((ID.%text^"CONSTRUCT"):"exception")}

 # id_list.action_code == "timer"
  -> {((ID.%text^"CONSTRUCT"):"timer")}

 # id_list.action_code == "co_output"
  -> {([id_list.opid,"_",ID.%text,"OUTPUT"]:"conditional")}

 # {(?:string:"")}
);

id_list.ucond_output =
((id_list.in_env(id_list.opid^"_"^ID.%text^"OUTPUT")<>"conditional")
 && (id_list.action_code == "output")
 -> [id_list.in_env(id_list.opid^"PARENT"),"_SPEC.DS",ID.%text,
    ".WRITE(",ID.%text,");\n"]
 # ""
);
```

```
        }
  ;



time
  : INTEGER_LITERAL unit
    { time.trn = ""; }
  ;

unit
  : MS
        { unit.value = 1;
        }

  | SEC
        { unit.value = 1000;
        }

  | MIN
        { unit.value = 60000;
        }

  | HOURS
        { unit.value = 3600000;
        }
  ;

reqmts_trace
  : BY id_list
    { reqmts_trace.trn = "";
        id_list.in_env = {(?:string:"")};
        id_list.action_code = "";
        id_list.tname = "";
        id_list.opid = "";
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
        }
```

```
 |
   { reqmts_trace.trn = ""; }
     ;

functionality
 : keywords informal_desc formal_desc
   { functionality.trn = ""; }
 ;

keywords
 : KEYWORDS id_list
   { keywords.trn = "";
        id_list.in_env = {(?:string:"")};
        id_list.action_code = "";
        id_list.tname = "";
        id_list.opid = "";
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
       }
 |
   { keywords.trn = ""; }
 ;

informal_desc
 : DESCRIPTION TEXT
   { informal_desc.trn = "\n"; }
 |
   {informal_desc.trn = ""; }
 ;

formal_desc
 : AXIOMS TEXT
   { formal_desc.trn = "\n"; }
 |
   { formal_desc.trn = ""; }
 ;

type_impl
```

```
 : IMPLEMENTATION ADA ID
   { type_impl.trn = ["procedure ",ID.%text," is;\n"]; }
 | IMPLEMENTATION type_name op_impl_0_list END
   { type_impl.trn = ["\n package DATA_TYPES is \n",type_name.trn,"\n",
      op_impl_0_list.trn,"\n",
      "end;\n"]; }
 ;

op_impl_0_list
 : op_impl_0_list OPERATOR  ID operator_impl
   { op_impl_0_list[1].trn = "";
        operator_impl.opid = ID.%text; }
 |
   { op_impl_0_list[1].trn = ""; }
 ;

operator_impl
 : IMPLEMENTATION ADA ID
   { operator_impl.trn = "";
        operator_impl.loc_ds_decl = "";
        operator_impl.timer_decl = "";
        operator_impl.out_env = {((ID.%text^"CONSTRUCT"):"atomic_operator")};
       }

 | IMPLEMENTATION psdl_impl
   { operator_impl.trn = psdl_impl.trn;
        operator_impl.loc_ds_decl = psdl_impl.loc_ds_decl;
        operator_impl.timer_decl = psdl_impl.timer_decl;
        psdl_impl.parent = operator_impl.opid;
        psdl_impl.in_env = operator_impl.in_env;
        psdl_impl.uncond_output_map = operator_impl.uncond_output_map;
        operator_impl.out_env =
         {((operator_impl.opid^"CONSTRUCT"):"composite_operator")} +|
           psdl_impl.out_env;
       }
 ;

psdl_impl
```

```
 : data_flow_diagram streams timers control_constraints informal_desc END
  { psdl_impl.trn = control_constraints.trn;
       psdl_impl.out_env =  streams.out_env +| control_constraints.out_env;
       psdl_impl.loc_ds_decl = streams.trn;
       psdl_impl.timer_decl = timers.trn;
       data_flow_diagram.in_env = psdl_impl.in_env;
       streams.in_env = psdl_impl.in_env;
       control_constraints.parent = psdl_impl.parent;
       control_constraints.in_env = psdl_impl.in_env;
       control_constraints.decl_map = data_flow_diagram.decl_map;
       control_constraints.uncond_output_map = psdl_impl.uncond_output_map;
      }
    ;

data_flow_diagram
 : GRAPH link_0_list
  { data_flow_diagram.trn = "";
       data_flow_diagram.decl_map = link_0_list.out_decls;
       link_0_list.in_decls = {(?:string:"")};
       link_0_list.in_env = data_flow_diagram.in_env;
      }
 ;

link_0_list
 : link link_0_list
  { link_0_list[1].trn = "";
       link_0_list[1].out_decls = link_0_list[2].out_decls;
       link_0_list[2].in_decls = link.out_decls;
       link.in_decls = link_0_list[1].in_decls;
       link.in_env = link_0_list[1].in_env;
       link_0_list[2].in_env = link_0_list[1].in_env;
       }
 |
  { link_0_list.trn = "";
       link_0_list.out_decls = link_0_list.in_decls;
      }
 ;
```

```
link
 : ID '.' ID opt_time ARROW ID
  { link.trn = "";
      link.out_decls =
        (link.in_decls(ID[3].%text^ID[1].%text^"READ") == "dup"
         -> {(?:string:"")}
         # {((ID[3].%text^"READ"):[link.in_decls(ID[3].%text^"READ"),

link.in_env("PARENT"),"_SPEC.DS",ID[1].%text,".READ(",ID[1].%text,
            ");\n"]) ((ID[3].%text^ID[1].%text^"READ"):"dup")}
        ) +|

        (link.in_decls([ID[2].%text,"_",ID[1].%text]) == "dup"
         -> {(?:string:"")}
         # {(ID[2].%text:[link.in_decls(ID[2].%text),
           ID[1].%text," : ",link.in_env(ID[1].%text^"TYPE"),
           ";\n"]) ([ID[2].%text,"_",ID[1].%text]:"dup")}
        ) +|

        (link.in_decls([ID[3].%text,"_",ID[1].%text]) == "dup"
         -> {(?:string:"")}
         # {(ID[3].%text:[link.in_decls(ID[3].%text),
           ID[1].%text," : ",link.in_env(ID[1].%text^"TYPE"),
           ";\n"]) ([ID[3].%text,"_",ID[1].%text]:"dup")}
        ) +| link.in_decls;
      }
 ;

opt_time
 : ':' time
  { opt_time.trn = ""; }
 |
  { opt_time.trn = "\n"; }
 ;


streams
 : DATA_STREAM type_decl
```

```
    { streams.trn = type_decl.trn;
        streams.out_env = type_decl.out_env;
        type_decl.opid = "";
        type_decl.action_code = "stream";
        type_decl.in_env = streams.in_env;
      }
  |
    {streams.trn = "";
      streams.out_env = {(?:string:"")};
      }
  ;

type_name
  : ID '[' type_decl ']'
    { type_name.trn = [ID.%text,"[",type_decl.trn,"]\n"];
      type_decl.opid = "";
      type_decl.action_code = "tname"; }
  | ID
    { type_name.trn = ID.%text; }
  ;

timers
  : TIMER id_list
    { timers.trn = [id_list.trn," : PSDL_TIMER;\n"];
      id_list.in_env = {(?:string:"")};
      id_list.action_code = "timer";
      id_list.tname = "";
      id_list.opid = "";
      id_list.count = 1;
      id_list.exp_env = {(?:int:"")};
     }

  |
    {timers.trn = "";
     }
    ;

control_constraints
```

```
: CONTROL
  { control_constraints.trn = "";
       control_constraints.out_env = {(?:string:"")};
     }

| CONTROL OPERATOR ID opt_trig opt_per opt_fin_w constraint_options
     more_constraints
  {control_constraints.trn =
       (control_constraints.in_env(ID.%text^"CONSTRUCT")
          == "composite_operator"
        -> ["procedure ",control_constraints.in_env("PARENT"),"_",
          ID.%text," is\n  begin\n     null;\n  end ",
          control_constraints.in_env("PARENT"),"_",ID.%text,";\n"]

        # ["procedure ",control_constraints.in_env("PARENT"),
          "_",ID.%text," is\n",control_constraints.decl_map(ID.%text),
          "\nbegin\n",opt_trig.streams_check,
          control_constraints.decl_map(ID.%text^"READ"),
          opt_trig.pred,
          (control_constraints.in_env(ID.%text^"PROCCALL") == ""
           -> [ID.%text,";\n"]
           # [ID.%text,"(",
              control_constraints.in_env(ID.%text^"PROCCALL") ,");\n"]
          ),
          constraint_options.trn,"\n",
          control_constraints.uncond_output_map(ID.%text),
          opt_trig.end_if_pred,opt_trig.end_if_streams,
          "end ",control_constraints.in_env("PARENT"),
          "_",ID.%text,";\n"]
       ) ^ more_constraints.trn;

       opt_trig.in_env = control_constraints.in_env;
       constraint_options.in_env = control_constraints.in_env;
       constraint_options.opid = ID.%text;
       control_constraints.out_env =
          {((ID.%text^"PARENT"):control_constraints.parent)}
          +| opt_trig.out_env
          +| constraint_options.out_env
```

```
        +| more_constraints.out_env;

      more_constraints.parent = control_constraints.parent;
      more_constraints.in_env = control_constraints.in_env;
      more_constraints.uncond_output_map =
        control_constraints.uncond_output_map;
      more_constraints.decl_map = control_constraints.decl_map;
      }

    |
      {control_constraints.trn = "";
      control_constraints.out_env = {(?:string:"")};
      }
  ;

more_constraints
    : OPERATOR ID opt_trig opt_per opt_fin_w constraint_options
      more_constraints
      {more_constraints[1].trn =
       (more_constraints.in_env(ID.%text^"CONSTRUCT") ==
         "composite_operator"
       -> ["procedure ",more_constraints[1].in_env("PARENT"),"_",
          ID.%text," is\n  begin\n    null;\n  end ",
          more_constraints[1].in_env("PARENT"),"_",ID.%text,";\n"]

       # ["procedure ",more_constraints[1].in_env("PARENT"),
          "_",ID.%text," is\n",more_constraints[1].decl_map(ID.%text),
          "\nbegin\n",opt_trig.streams_check,
          more_constraints.decl_map(ID.%text^"READ"),
          opt_trig.pred,
          (more_constraints[1].in_env(ID.%text^"PROCCALL") == ""
           -> [ID.%text,";\n"]
           # [ID.%text,"(",
             more_constraints[1].in_env(ID.%text^"PROCCALL"),");\n"]
          ),
          constraint_options.trn,"\n",
          more_constraints[1].uncond_output_map(ID.%text),
          opt_trig.end_if_pred,opt_trig.end_if_streams,
```

```
        "end ",more_constraints[1].in_env("PARENT"),
        "_",ID.%text,";\n"]
      ) ^ more_constraints[2].trn;

      opt_trig.in_env = more_constraints.in_env;
      constraint_options.in_env = more_constraints.in_env;
      constraint_options.opid = ID.%text;
      more_constraints[1].out_env =
        {((ID.%text^"PARENT"):more_constraints.parent)} +|
        opt_trig.out_env +|
        constraint_options.out_env +|
        more_constraints[2].out_env;
      more_constraints[2].in_env = more_constraints[1].in_env;
      more_constraints[2].uncond_output_map =
        more_constraints[1].uncond_output_map;
      more_constraints[2].decl_map = more_constraints[1].decl_map;

      }
     |
      {more_constraints.trn = "";
      more_constraints.out_env = {(?:string:"")} ;
      }
    ;


constraint_options
    : OUTPUT id_list IF predicate reqmts_trace constraint_options
      { constraint_options[1].trn =
        ["if ",predicate.trn,"\nthen\n    ",id_list.trn,"\nend if;\n",
         constraint_options[2].trn] ;

       constraint_options[2].opid = constraint_options[1].opid;
       constraint_options[2].in_env = constraint_options[1].in_env;
       constraint_options[1].out_env = id_list.out_env +|
                          constraint_options[2].out_env;
       predicate.in_env = constraint_options[1].in_env;
       id_list.in_env = constraint_options[1].in_env;
       id_list.action_code = "co_output";
```

```
        id_list.tname = "";
        id_list.opid = constraint_options.opid;
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
       }

      | EXCEPTION ID opt_if_predicate reqmts_trace constraint_options
       { constraint_options[1].trn = constraint_options[2].trn;
        constraint_options[1].out_env = constraint_options[2].out_env;
        constraint_options[2].opid = constraint_options[1].opid;
        constraint_options[2].in_env = constraint_options[1].in_env;
        opt_if_predicate.in_env = constraint_options[1].in_env;
       }

      | timer_op ID  opt_if_predicate reqmts_trace constraint_options
       { constraint_options[1].trn =
          [opt_if_predicate.if,timer_op.trn,"(",
           constraint_options[1].in_env("PARENT"),"_SPEC.TL",ID.%text,
           ");\n",opt_if_predicate.end_if,constraint_options[2].trn];
        constraint_options[1].out_env = constraint_options[2].out_env;
        constraint_options[2].opid = constraint_options[1].opid;
        constraint_options[2].in_env = constraint_options[1].in_env;
        opt_if_predicate.in_env = constraint_options[1].in_env;
       }

      |
       { constraint_options.trn = "";
        constraint_options.out_env = {(?:string:"")};
       }
      ;

opt_trig
 : TRIGGERED trigger opt_if_predicate reqmts_trace
      { opt_trig.out_env = trigger.out_env;
        opt_trig.pred = opt_if_predicate.if;
        opt_trig.end_if_pred = opt_if_predicate.end_if;
        opt_trig.streams_check = trigger.if;
        opt_trig.end_if_streams = trigger.end_if;
```

```
        trigger.in_env = opt_trig.in_env;
        opt_if_predicate.in_env = opt_trig.in_env;
       }
|
       {opt_trig.out_env = {(?:string:"")};
        opt_trig.pred = "";
        opt_trig.end_if_pred = "";
        opt_trig.streams_check = "";
        opt_trig.end_if_streams = "";
       }
;

trigger
 : ALL id_list
  { trigger.if = ["if ",id_list.trn,"\nthen\n"];
        trigger.end_if = "end if;\n";
        trigger.out_env = id_list.out_env;
        id_list.action_code = "by_all";
        id_list.tname = "";
        id_list.opid = trigger.in_env("PARENT");
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
       }

 | SOME id_list
  { trigger.if = ["if ",id_list.trn,"\nthen\n"];
        trigger.end_if = "end if;\n";
        trigger.out_env = id_list.out_env;
        id_list.action_code = "by_some";
        id_list.tname = "";
        id_list.opid = trigger.in_env("PARENT");
        id_list.count = 1;
        id_list.exp_env = {(?:int:"")};
       }

 |
  {trigger.if = "";
        trigger.end_if = "";
```

```
        trigger.out_env = {(?:string:"")};
       }
 ;

opt_per
 :PERIOD time reqmts_trace
  { opt_per.trn = "\n"; }
 |
  { opt_per.trn = ""; }
 ;

opt_fin_w
 : FINISH time reqmts_trace
  { opt_fin_w.trn = "\n"; }
 |
  { opt_fin_w.trn = ""; }
 ;



timer_op
 : READ
       { timer_op.trn = "PSDL_TIMER.READ"; }
    | RESET
  { timer_op.trn = "PSDL_TIMER.RESET"; }
 | START
  { timer_op.trn = "PSDL_TIMER.START"; }
 | STOP
  { timer_op.trn = "PSDL_TIMER.STOP"; }
 ;

opt_if_predicate
 : IF predicate
  { opt_if_predicate.if = ["if ",predicate.trn,"\nthen\n"];
       opt_if_predicate.end_if = "end if;\n";
       predicate.in_env = opt_if_predicate.in_env;
      }
```

```
    |
      {opt_if_predicate.if = "";
       opt_if_predicate.end_if = "";
      }
 ;


expression_list
 : expression
   {expression_list.trn = expression.trn;
       expression_list.exp_env = {(expression_list.count:expression.trn)
                          (0:i2s(expression_list.count))
                          (?:int:"")}; }

    | expression ',' expression_list
   {expression_list[1].trn =
   [expression.trn,",",expression_list[2].trn];
       expression_list[1].exp_env =
       {(expression_list[1].count:expression.trn)} +|
       expression_list[2].exp_env;
       expression_list[2].count = expression_list[1].count + 1; }
 ;

expression
     : INTEGER_LITERAL
       {expression.trn = INTEGER_LITERAL.%text; }
     | REAL_LITERAL
       {expression.trn = REAL_LITERAL.%text; }
     | STRING_LITERAL
       {expression.trn = STRING_LITERAL.%text; }
     | TRUE
       {expression.trn = " true "; }
     | FALSE
       {expression.trn = " false "; }
     | ID
       {expression.trn = ID.%text; }
     | type_name '.' ID '(' expression_list ')'
       {expression.trn = [type_name.trn,ID.%text,"(",expression_list.trn,
```

```
                    ") "];
        expression_list.count = 1; }
    ;
predicate
    : relation
      {predicate.trn = relation.trn;
       predicate.type = relation.type;
       relation.in_env = predicate.in_env;
      }

    | relation AND predicate
      {predicate[1].trn = [relation.trn," and ",predicate[2].trn];
       predicate[1].type = "";
       predicate[2].in_env = predicate[1].in_env;
       relation.in_env = predicate[1].in_env;
      }

    | relation OR predicate
      {predicate[1].trn = [relation.trn," or ",predicate[2].trn];
       predicate[1].type = "";
       predicate[2].in_env = predicate[1].in_env;
       relation.in_env = predicate[1].in_env;
      }
    ;


relation
    : simple_expression rel_op simple_expression
      {relation.trn = rel_op.trn;
       simple_expression[1].in_env = relation.in_env;
       simple_expression[2].in_env = relation.in_env;
       relation.type =
        (simple_expression[1].type == "timer" ||
         simple_expression[2].type == "timer"
         -> "timer"
         # simple_expression[1].type == "excp" ||
           simple_expression[2].type == "excp"
           -> "excp"
```

```
        # ""
       );
      rel_op.left_op = simple_expression[1].trn;
      rel_op.right_op = simple_expression[2].trn;
      rel_op.parent = relation.in_env("PARENT");
      rel_op.opn_type =
       (simple_expression[1].type == "timer" ||
        simple_expression[2].type == "timer"
        -> "timer_op"
        # "arithmetic"
       );
     }

   | simple_expression
     {relation.trn = simple_expression.trn;
      relation.type = simple_expression.type;
      simple_expression.in_env = relation.in_env;
     }
   ;

simple_expression
   : INTEGER_LITERAL unit
     {simple_expression.trn = i2s(s2i(INTEGER_LITERAL.%text)
                  * unit.value);
      simple_expression.type = "timer";
     }
   | sign INTEGER_LITERAL
     {simple_expression.trn = [sign.trn,INTEGER_LITERAL.%text];
      simple_expression.type = "";
     }
   | sign REAL_LITERAL
     {simple_expression.trn = [sign.trn,REAL_LITERAL.%text];
      simple_expression.type = "";
     }
   | ID
     {simple_expression.trn = ID.%text;
      simple_expression.type =
       (simple_expression.in_env(ID.%text^"CONSTRUCT") == "timer"
```

```
    -> ""
    # simple_expression.in_env(ID.%text^"CONSTRUCT")
   );
  }
 | STRING_LITERAL
  {simple_expression.trn = STRING_LITERAL.%text;
   simple_expression.type = "";
  }
 | '(' predicate ')'
  {simple_expression.trn = ["(",predicate.trn,")"];
   simple_expression.type = predicate.type;
   predicate.in_env = simple_expression.in_env;
  }
 | NOT ID
  {simple_expression.trn = ["not ",ID.%text];
   simple_expression.type = "";
  }
 | NOT '(' predicate ')'
  {simple_expression.trn = ["not (",predicate.trn,")"];
   simple_expression.type = "";
   predicate.in_env = simple_expression.in_env;
  }
 | TRUE
  {simple_expression.trn = " true ";
   simple_expression.type = "";
  }
 | FALSE
  {simple_expression.trn = " false ";
   simple_expression.type = "";
  }
 | NOT TRUE
  {simple_expression.trn = [" not true "];
   simple_expression.type = "";
  }
 | NOT FALSE
  {simple_expression.trn = " not false ";
   simple_expression.type = "";
  }
```

```
    ;

rel_op
 : '<'
   {rel_op.trn =
        (rel_op.opn_type == "timer_op"
         -> ["PSDL_TIMER.\"<\"(",rel_op.left_op,",",rel_op.right_op,") "]
         # [rel_op.left_op," < ",rel_op.right_op]
        );
       }
   | '>'
   {rel_op.trn =
        (rel_op.opn_type == "timer_op"
         -> ["PSDL_TIMER.\">\"(",rel_op.left_op,",",rel_op.right_op,") "]
         # [rel_op.left_op," > ",rel_op.right_op]
        );
       }

   | '='
   {rel_op.trn =
        (rel_op.opn_type == "timer_op"
         -> ["PSDL_TIMER.\"=\"(",rel_op.left_op,",",rel_op.right_op,") "]
         # [rel_op.left_op," = ",rel_op.right_op]
        );
       }

   | GTE
   {rel_op.trn =
        (rel_op.opn_type == "timer_op"
         -> ["PSDL_TIMER.\">=\"(",rel_op.left_op,",",rel_op.right_op,") "]
         # [rel_op.left_op," >= ",rel_op.right_op]
        );
       }

   | LTE
   {rel_op.trn =
        (rel_op.opn_type == "timer_op"
         -> ["PSDL_TIMER.\"<=\"(",rel_op.left_op,",",rel_op.right_op,") "]
```

```
        # [rel_op.left_op," <= ",rel_op.right_op]
       );
      }

    | NEQV
    {rel_op.trn =
       (rel_op.opn_type == "timer_op"
        -> ["PSDL_TIMER.\"/=\"(",rel_op.left_op,",",rel_op.right_op,") "]
        # [rel_op.left_op," /= ",rel_op.right_op]
       );
      }

    | ':'
      {rel_op.trn =
       (rel_op.right_op == "NORMAL"
        -> [rel_op.parent,"_SPEC.DS",rel_op.left_op,".IS_NORMAL "]
        # [rel_op.parent,"_SPEC.DS",rel_op.left_op,
           ".IS_EXCEPTION(",rel_op.right_op,") "]
       );
      }
    ;

sign
    : '+'
      {sign.trn = "+ "; }
    | '-'
      {sign.trn = "- "; }
    |
      {sign.trn = ""; }
    ;
```

APPENDIX E. TRANSLATION TEMPLATES

A. TL PACKAGE TEMPLATE

```
with PSDL_SYSTEM;
use PSDL_SYSTEM;
package TL is
```

```
type PSDL_EXCEPTION is (psdl_excp1, psdl_excp2, ..., psdl_excpN);
excp1, excp2, ..., excpN  : exception;
```

```
procedure Atomic_Driver1;
procedure Atomic_Driver2;
...
procedure Atomic_DriverJ;
```

```
end TL;

package body TL is
```

```
Atomic procedures drawn from software base.
```

```
PSDL operator specification packages.
```

```
PSDL atomic operator driver procedures.
```

```
end TL;
```

B. PSDL OPERATOR SPECIFICATION PACKAGE TEMPLATE

This package template guides the translation of PSDL specifications into Ada packages. The packages contain only the data stream and timer instantiations that are local to Current_Operator. Current_Operator is the name listed in the header of the PSDL specification. Specification packages are produced only for composite operators.

```
with Parent_Operator_SPEC;      -- Current_Operator's parent.
use Parent_Operator_SPEC;
package Current_Operator_SPEC is
```

INPUT data stream instantiations.

OUTPUT data stream instantiations.

STATES data stream instantiations.

DATA STREAM data stream instantiations.

TIMER instantiations.

```
end Current_Operator_SPEC;
```

C. PSDL ATOMIC OPERATOR DRIVER PROCEDURE TEMPLATE

This template guides the construction of the procedure that is used to implement an atomic PSDL operator. It implements the Ada code necessary to simulate all of the control constraints on the operator. It also contains a procedure call to the code which implements the actual operator drawn from the software base. When implemented, it is these procedures that will serve as the interface between the static and dynamic schedules and the operators from the software base.

```
with Current_Operator_SPEC;
procedure Parent_Operator_Current_Operator is

   Data stream variable declarations.

begin
      if Data_Trigger then
         Read input data streams.
         if  Trigger_Condition then

            Current_Operator (Paramenter_List);  -- Atomic procedure call

            Control constraint implementations.
               Timer operations,
               exception operations,
               conditional output operations

            Unconditional output statements.

      end if;
   end if;
```

```
exception
    when   Ada_Exceptions =>
        Write to output exception streams.
    when others  =>
        raise;      -- re-raise Ada exception if a corresponding PSDL exception not
                    -- declared
```

end Parent_Operator_Current_Operator;

LIST OF REFERENCES

1. Berzins, V., and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development Using Ada*, Addison-Wesley, 1988.

2. Whitten, J. L., Bentley, L. D., and Ho, T. I. M., *Systems Analysis and Design Methods*, Times Mirror/Mosby College, St. Louis, Missouri,1986.

3. Luqi, and Ketabchi, M., "A Computer Aided Prototyping System," *IEEE Software*, v. 5, pp. 66-72, March 1988.

4. Porter, S. W., *Design of a Syntax Directed Editor for PSDL*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

5. Thorstenson, R. K., *A Graphical Editor for the Computer Aided Prototyping System (CAPS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

6. Luqi, *Rapid Prototyping for Large Software System Design*, Ph.D. Dissertation, University of Minnesota, Minneapolis, Minnesota, May 1986.

7. Galik, D., *A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System*, Master's thesis, Naval Postgraduate School, Monterey, California, December 1988.

8. Marlowe, L. C., *A Scheduler for Critical Timing Constraints*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

9. Wood, M. B., *An Execution Support System for Rapid Prototyping*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

10. Janson, D., *A Static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

11. Herndon, R. M., *Automatic Construction of Language Translators*, Ph.D. Dissertation, University of Minnesota, Minneapolis, Minnesota, May 1987.

12. Knuth, D. E., *Semantics of Context-Free Languages*, Mathematical Systems Theory, pp. 127-145, November 1967.

13. Luqi, Berzins, V., and Yeh, R., *A Prototyping Language for Real-Time Software*, IEEE Transactions on Software Engineering, pp. 1409-1423, October 1988.

14. Berzins, V., and Luqi, "Semantics of a Real-Time Language," paper presented at the Real-Time Systems Symposium, Huntsville, Alabama, 29 November 1988.

15. Luqi, Berzins, V., "Execution of a High Level Real-Time Language,"paper presented at the Real-Time Systems Symposium, Huntsville, Alabama, 29 November 1988.

16. Raum, H. G., *The Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

16. Moffitt, C., *A Language Translator for a Computer Aided Rapid Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1988.

17. United States Department of Defense, Under Secretary for Research and Engineering, *Reference Manual for the Ada Programming Language*, Government Printing Office,Washington, DC, 1983.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, Virginia 22217-5000	1
4.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, D.C. 20363-5100	1
5.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, D.C. 20301	1
6.	Naval Sea Systems Command Attn. CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 20363-5100	1
7.	Office of the Secretary of Defense Attn. CDR Barber STARS Program Office Washington, D.C. 20301	1
8.	Office of the Secretary of Defense Attn. Mr. Joel Trimble STARS Program Office Washington, D.C. 20301	1
9.	Commanding Officer Naval Research Laboratory Code 5150 Attn. Dr. Elizabeth Wald Washington, D.C. 20375-5000	1

10. Navy Ocean System Center — 1
Attn. Linwood Sutton, Code 423
San Diego, California 92152-5000

11. National Science Foundation — 1
Attn. Dr. William Wolf
Washington, D.C. 20550

12. National Science Foundation — 1
Division of Computer and Computation Research
Attn. Dr. Peter Freeman
Washington, D.C. 20550

13. National Science Foundation — 1
Director, PYI Program
Attn. Dr. C. Tan
Washington, D.C. 20550

14. Office of Naval Research — 1
Computer Science Division, Code 1133
Attn. Dr. Van Tilborg
800 N. Quincy Street
Arlington, Virginia 22217-5000

15. Office of Naval Research — 1
Applied Mathematics and Computer Science, Code 1211
Attn. Mr. J. Smith
800 N. Quincy Street
Arlington, Virginia 22217-5000

16. Defense Advanced Research Projects Agency (DARPA) — 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Jacob Schwartz
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

17. Defense Advanced Research Projects Agency (DARPA) — 1
Integrated Strategic Technology Office (ISTO)
Attn. Dr. Squires
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

18. Defense Advanced Research Projects Agency (DARPA) — 1
Integrated Strategic Technology Office (ISTO)
Attn. MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

19. Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

20. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

21. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

22. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

23. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

24. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

25. U.S Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, New York 13441-5700

26. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. William E. Rzepka
Griffis Air Force Base, New York 13441-5700

27. Professor Luqi 1
Code 52LQ
Naval Postgraduate School
Computer Science Department
Monterey, California 93943-5100

28. LT Charles E. Altizer 1
Combat
USS Coral Sea (CV-43)
FPO New York, New York 09550-2720